BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

PATENT
5800-01700

"EXPRESS MAIL" MAILING LABEL NUMBER EV324269705US
DATE OF DEPOSIT MAY 11, 2004
I HEREBY CERTIFY THAT THIS PAPER OR FEE IS BEING DEPOSITED WITH THE UNITED STATES POSTAL SERVICE "EXPRESS MAIL POST OFFICE TO ADDRESSEE" SERVICE UNDER 37 C.F.R. §1.10 ON THE DATE INDICATED ABOVE AND IS ADDRESSED TO THE COMMISSIONER FOR PATENTS, BOX PATENT APPLICATION, WASHINGTON, D.C. 20231

Derrick Brown

SECURE VM SUPPORT

By:

Geoffrey S. Strongin
Kevin J. McGrath
Alexander Klaiber
David S. Christie

B. Noël Kivlin
Meyertons, Hood, Kivlin, Kowert & Goetzel, P.C.
P.O. Box 398
Austin, TX 78767-0398

Secure VM Support

Spec & Usage

Abstract

This document describes the detailed operation of the "basic" AMD Secure-VM support, and outlines how to use those mechanisms for full virtualization and security of x86 systems. The proposed hardware mechanisms should be usable for efficient para-virtualization as well. The overall VM architecture is crafted such as to allow clean extensions to the basic functionality.

Release 0.1

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Terminology and Goals

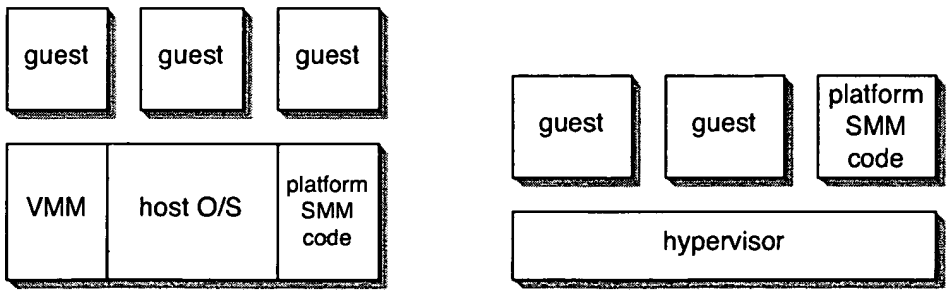
We plan to support at least two models of VM use:

- **Full virtualization**: a VMM (either integrated with a full-fledged operating system, or standalone as a "hypervisor") running multiple unmodified guests. Each guest sees contiguous, zero-based memory; the VMM uses shadow page tables to remap memory appropriately. Since guest physical addresses are remapped, guests are not permitted to access I/O devices directly (there being no Northbridge mechanism to similarly remap device physical addresses).
- **Para-virtualization**: the VMM supports one or more VM-aware guests; the guests negotiate with the VMM for physical memory which is non-contiguous. No remapping of guest physical addresses is performed; the VMM relies on either hardware permission-checking mechanisms or page table editing to ensure guests only access "their" physical memory. Since there is no address remapping, guests can be permitted to access devices directly. The Northbridge uses bitvectors indexed by physical page number to restrict device access to selected memory regions. In fact, the Northbridge will contain (pointers to) multiple such tables, one for each of several *protection domains*. Devices are mapped to protection domains based on their HT bus/device ID — among other things, this allows multiple guests to have direct access to (disjoint) sets of devices.

We intend to support both fully and para-virtualized guests coexisting in the same system.

VM systems differ along another axis: hosted versus non-hosted. In a hosted VM system, the VMM coexists with, and relies on, a full-fledged host operating system. The VMM typically trusts the host O/S and any device drivers it contains (as well as any platform SMM code set up by the platform's BIOS). In contrast, more security-conscious VM systems may choose to implement the VMM (typically called a hypervisor in this case, though we use the terms interchangeably) as a very thin layer that is as free as possible of device drivers; in such a system, the VMM may not even fully trust the platform's SMM code. We support the needs of either architecture.

Figure 1. Hosted versus Non-Hosted VM Architecture



1.2 Outline

Section 2 and Section 3 highlight some key features of the proposed hardware support and the intended use by a virtual machine monitor, respectively.

Section 4 presents the details of the VM hardware support; the use by software (i.e., a VMM or hypervisor) of the hardware support is discussed in more detail in Section 5.

2. Basic VM Hardware Support, Overview

In this section, we briefly outline the major components of the proposed hardware support; more details are given in Section 4. Very broadly speaking, h/w support falls into two categories: virtualization support and security support (though the two are of course complementary).

2.1 Virtualization Support

The key hardware support for efficient x86 virtualization consists of mechanisms for fast *world switch* between VMM and guest, the ability to *intercept* selected instructions or events in the guest, and external (DMA) *access protection* for memory. Additional *VM assists* for interrupt handling, *virtual interrupt* support and *APIC.TPR* virtualization, as well as a guest/host *tagged TLB* should drastically reduce virtualization overhead.

2.1.1 Guest Mode

We introduce a new processor mode, *guest mode*. Guest mode can only be entered through a new instruction, SVM ("Start Virtual Machine")[1]. When in guest mode, the behavior of some x86 instructions changes to facilitate virtualization. We envision that in an implementation, there will not actually be a single "guest mode" bit — rather, there will be several bits, each modifying the behavior of one or more operations/instructions.

2.1.2 SVM Instruction

The SVM instruction is the cornerstone of the proposed VM hardware support. It takes as 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 SVM instruction saves a subset of host processor state, loads the corresponding guest state from the VMCB and runs the guest code until one of several *intercept* events (specified in the VMCB) occurs. At this point, the processor stores the guest state back into the VMCB, reloads the host state, and continues execution of host code at the instruction following the SVM instruction. An entry in the VMCB will indicate to the VMM the reason why guest execution was suspended. Principally, the VMCB contains

- a list of which instructions or events (e.g., write to `cr3`) in the guest to intercept;
- various control bits for further specifying the execution environment of the guest (such as different modes for interrupt handling), or for indicating special actions to be taken before running guest code (such as flushing the TLB of guest entries);
- guest processor state (such as control registers etc.).

The SVM instruction is designed to simplify the addition future extensions and assists.

1. The SVM instruction is similar in spirit to the 1970s-era "SIE" instruction found in IBM mainframes of the day.

2.1.3 External Access Protection

Since we want to give para-virtualized guests (which use machine physical addresses) direct access to (selected) I/O devices, we need a mechanism to prevent devices "owned" by one guest from accessing any memory owned by another guest (or the hypervisor). To do so, we use the following hardware support:

- any request from a device is tagged with an identifier for that device (deviceID). Initially, we will use HyperTransport bus/device IDs (which gives us some limited ability to assign device ownership). Future implementations will likely use PciExpress deviceIDs.
- a table in the NB (set up by the VMM) maps from deviceID to a protection domain. Assuming that HT bus IDs are small enough, this can be done with a fixed set of registers.
- associated with each protection domain is a bitvector with one bit per 4KB page (and indexed by physical page number) that indicates whether a given page of physical memory is accessible by that domain. At least initially, it seems sufficient to support a fixed number of 8–16 protection domains in the NB, so again we can use a fixed set of registers. The bitvector is essentially the DEV (Device Exclusion Vector) table described in the old SEM documentation.
- the NB will need a TLB for caching parts of the DEV; that TLB should be tagged by the protection domain (or deviceID). Otherwise, every device access to memory would require an extra memory read for checking protections.

IMPORTANT: to guarantee security, the platform must allow the VMM to prevent peer-to-peer transactions between devices (at least between devices belonging to different protection domains). Details on how this is done are beyons the scope of this document.

2.1.4 Tagged TLB

We propose that for better protection and flexibility, the VMM be mapped in a different address space than the guest[1]; in order to reduce the cost of world switches, we tag the TLB with an address space identifier (ASID) distinguishing host-space entries from guest-space entries. The VMCB contains a field indicating the ASID to use while executing the guest. As a minimum, we will require a single ASID bit: host versus guest. It is desirable to provide more bits, so the translations from several guests can coexist in the TLB.

In typical use, entering and exiting a guest (via SVM) will involve writing `cr3`. However, unlike ordinary `cr3` writes, the TLB will not be flushed when doing so; the processor will merely switch ASIDs. If the VMM needs to flush the TLB of guest entries, it does so by explicitly requesting that operation in the VMCB.[2]

1. Many existing virtualization schemes map part of the VMM into the guest address space, in order to eliminate the cost of address space switching. This has various drawbacks, and typically requires use of a separate mechanism to protect the VMM from guest access.
2. A SVM -initiated flush of the TLB shall flush all guest entries with matching ASID, regardless of whether they are marked "global" or not.

2.1.5 Interrupt and APIC Support

To facilitate reasonably efficient virtualization of interrupts, we include the following support

- prevent guests from masking physical INTR (maskable) interrupts via the `eflags` register;
- physical interrupts cause a running guest to exit, so the VMM may process the interrupt;
- the VMM can indicate (via bits in the VMCB) whether a "virtual" maskable interrupt for a guest is pending, as well as that interrupt's vector and priority;
- hardware supplies guests with their own "virtual" APIC task priority register, which does not control the physical localAPIC, but instead determines whether a pending virtual interrupts is of high enough priority to be admitted;
- guests take virtual interrupts (when enabled and of high enough priority) just like a normal x86 would take physical interrupts, except that the vector is taken from a register (as opposed to running a INTAK cycle to the APIC)

Other guest interactions with its localAPIC will be intercepted by the VMM. In particular, the VMM tracks EOI (End-of-Interrupt) cycles to the virtual localAPIC. This allows the VMM to perform an EOI to the physical localAPIC on behalf of the guest, as well as (together with the fact that new physical interrupts also intercept to the VMM) maintain a s/w model of the guest's virtual APIC.

To guard against malicious guests that leave high-priority interrupts unacknowledged forever (and thus shut out other guest's interrupts), we need to add a register to the localAPIC that allows the VMM to mask any interrupt vector (until the guest issues its EOI). Note that this implies that a given interrupt vector cannot be shared among different guests.

Note: virtualization of the ioAPIC is a platform issue and not treated in this document; a move to MSI (Message Signalled Interrupts) may enable an ioAPIC free platform architecture.

2.1.6 Fix Non-Restartable Instructions

Some instructions on the x86 are not restartable; task switches being the canonical example. By adding another layer of address translation (e.g., through shadow page tables) and corresponding intercepts (exceptions), more (guest) instructions may become non-restartable. This is because the additional address translation will introduce page faults (or similar intercepts to the VMM) that a guest does not expect. For example, a guest O/S may think a page is pinned, yet the VMM has not yet filled in the shadow page table entry corresponding to that page. The VMM will have to restart the faulting instruction after the shadow page table has been filled in. There are instances other than task switches where we need to take great care in inserting SVM intercept checks in a manner that leaves the instruction restartable after an intercept.

2.2 Security Support

To guarantee secure initialization of the VMM/hypervisor, we need to retain the SKINIT instruction and system support from SEM essentially unchanged. Extra support is added to the hardware to allow the hypervisor to intercept transitions to SMM (System Management Mode), and run platform SMM code inside a guest (or "container").

3. Virtualization Software Overview

This section outlines how a hypervisor might use the VM hardware support. For details, the reader is referred to Section 5.

3.1 Memory Protection

The current proposal assumes that memory protection (on the CPU side) will be effected via software techniques, either through shadow page tables or in-place page table editing (the latter is essentially the method proposed in SEM). The guest must *always* run with paging enabled (which poses some problems for virtualizing real mode, see Section 5.4.5) and the VMM is responsible for maintaining proper shadow page tables.

3.2 Memory Map

We propose that the VMM and host not be mapped into the guest's address space; instead the VMM will be mapped in a different address space and hence entirely protected from accesses by the guest. We plan to support a TLB tagged with address space IDs, so as to avoid flushing the TLB when performing a world switch. There will be at least two address space IDs: "host" and "guest"; we may choose to support multiple guest ASIDs. The guest's physical system memory will presumably be mapped into the host address space, for quick access by the VMM.

3.3 Device Access

We propose that only para-virtualized guests that use real machine physical addresses be permitted to access devices directly; all other guests must use virtual devices. (Note that this supports the SEM scenario of a single guest; future hardware extensions could potentially allow direct device access even to fully virtualized guests.)

A hypervisor would enumerate all physical devices, and only make selected subsets of the devices visible (in PCI Config space) to guests. Virtual devices can be made visible in the same manner.

The prevent rogue DMA access by devices controlled by a (non-trusted) guest, the Northbridge uses a bitvector (like the SEM DEV table, covering the guest's physical DRAM space in 4KB chunks) to verify that a device is permitted to access a given page of physical memory. As an extension to what was supported in SEM, we propose that the Northbridge support multiple *protection domains*, each with its own DEV table. This lets us assign sets of devices to different guests, and protect guests from each others' devices. The Northbridge contains a table that maps HT bus/device IDs to protection domains; that table (and the per-domain DEV tables) must be maintained by the VMM.

Note: in future extensions of VM support, we will likely use 16-bit PciExpress device IDs in lieu of HT bus/device IDs; this will require some changes in the device-to-domain mapping structure, and thus the VMM software that maintains those maps.

3.4 Interrupts

Hardware support ensures that guest writes to `eflags.if` do not affect physical interrupts; instead `eflags.if` controls virtual interrupts which the VMM can raise via fields in the VMCB. Specifically, SVM indicates to a guest whether a virtual INTR interrupt is pending, and what its priority and vector are; when that interrupt is enabled in the guest (based on TPR, eflags.if, interrupt shadow), the guest takes the interrupt as though is was physical (though without an INTAK cycle), and without any involvement from the VMM.

The VMM maintains a software model of a localAPIC for each guest; it can do so because the arrival of physical interrupts and EOI operations from the guest cause an exit to the VMM. From the s/w model, the VMM can determine the highest-priority interrupt pending for a given guest and inject a corresponding virtual interrupt into the guest, via control bits in the VMCB.

Since some guests make very heavy use of the APIC's TPR register, we provide h/w support for virtualizing that register in the guest, without VMM involvement. Other guest interactions with the local APIC are trapped to the VMM (including an APIC EOI, at which point the VMM can update the s/w model of the guest's localAPIC).

As a possible optimization for settings with a *single* guest, SVM can specify whether the guest is allowed to process physical maskable interrupts (INTR), and control the (physical) localAPIC directly.

3.5 Security

Security is largely orthogonal; software can rely on the SKINIT instruction to start up a trusted hypervisor. There is sufficient h/w support to allow the hypervisor to intercept SMI and INIT events, and run platform SMM code inside a container.

4. Detailed Hardware Operation

This section describes the operation of the proposed hardware extensions. These extensions can loosely be grouped into the following categories

- state switch — SVM, VMSAVE, VMLOAD instructions, plus global interrupt flag (GIF) and instructions to manipulate the latter (STGI, CLGI). Sections 4.1.2, 4.3 and 4.5.
- intercepts — allow the VMM to intercept "sensitive" operations in the guest, without having to use traditional "ring compression" techniques. Sections 4.1.4 and 4.2.
- interrupt & APIC assists — physical interrupt intercepts, virtual interrupt support, APIC.TPR virtualization. Section 4.9.
- SMI and NMI intercepts and assists — Section 4.10.
- restartable microcode — Section 4.11.
- external (DMA) access protection — Section 4.12.
- security — SKINIT instruction, various control registers. Section 4.6.

4.1 SVM Instruction

The SVM has an implicit addressing mode of "[eax]"; eax is meant to contain the *physical* (!) address of a 4KB-aligned page, the Virtual Machine Control Block (VMCB), which describes a virtual machine (guest) to be executed. Principally, the VMCB contains

- the *intercept vector* (which instructions or events in the guest to intercept)
- a *state-save* area (where the guest state is saved)
- various control bits (detailed below)

4.1.1 Basic Operation

SVM is only available in CPL-0, and only in protected mode with paging enabled. It checks that the VMCB is 4KB aligned and writable (so on vm-exit, it can write back the new guest state) and saves the physical address of the VMCB (this reduces problems with the host TLB being out of sync with the page tables).

The SVM instruction saves some host processor state (in main memory, at the physical address specified via a new MSR: VM_HSAVE_AREA), then loads corresponding guest state from the VMCB's state-save area. It also reads additional control bits from the VMCB, which allow the VMM to flush the guest TLB, inject virtual interrupts into the guest, etc.

SVM then sanity-checks the guest state just loaded. If illegal state has been loaded (e.g., segment limits inconsistent with processor mode), the processor exits back to host mode (after restoring the host state).

Otherwise, the processor now runs the guest code until an *intercept* event occurs that the VMM is interested in (as specified in the VMCB's intercept vector). At this point, the processor stores the guest state back into the VMCB, reloads the host state, and continues execution of host code following the SVM instruction. An entry in the VMCB will indicate to the VMM the reason why guest execution was suspended. If the host state loaded upon exit from the guest is found to be inconsistent, the processor enters shutdown.

SVM is designed to save/restore just a near-minimal amount of state to allow a VMM to resume executing after a guest has exited, while ensuring that guest state does not affect execution of the host. This will allow the VMM to handle "simple" intercept conditions relatively quickly. If more guest state needs to be saved/restored (e.g., to handle more complex intercepts, or to switch to a different guest), the VMM can employ the VMSAVE and VMLOAD instructions.

4.1.2 State Switch

SVM saves the following host state at the physical address contained in the new MSR, `vm_hsave_pa`:

- `cs.sel, rip` — so host can resume running at intended address. **Important:** for `cs`, only save the selector.
- `rflags, rax` — processor mode and the register used by SVM to address the VMCB (the latter cannot be switched by the VMM software prior to the SVM).
- `ss.sel, rsp` — so on guest exit, the host has a place to resume from after a switch. For ss, save only the selector.
- `cr0, cr3, cr4, efer` — mainly to save the host paging mode.
- `idtr, gdtr` (the pseudo-descriptors).
- `es.sel` and `ds.sel` if the host is in 32-bit mode (`efer.lma==0`). Again, only save the segment selectors.

Note: on vm-exit, the above host state is reloaded, and `dr7` is set to the canned value of "all breakpoints disabled". If the host is in 32-bit mode, the `cs`, `ds`, and `es` selectors are reloaded from the save area, and then the descriptors are accordingly re-read from the descriptor tables. Any exception incurred while re-reading the segment descriptors causes the processor to shutdown. The reloaded host state is checked for consistency; any error causes the processor to shutdown.

After saving host state, SVM loads the following guest state from the VMCB:

- `cs.all, rip` — guest resumes execution at this address. **Important:** for cs, load the hidden state of the segment register, not just the selector. The guest CPL is inferred from the `cs` selector.
- `rflags, rax`.
- `ss.all, rsp`.
- `cr0, cr3, cr4, efer` — set up guest paging mode. **Important:** writing paging-related control registers via SVM (or on exit from SVM) should *not* flush the TLB (since we are switching to a

different ASID).

- a flag indicating whether the guest is in the shadow of an `sti` instruction. If set, the SVM skips the check for pending interrupts at the end of its microcode (i.e., just before executing the first guest instruction.
- `idtr`, `gdtr` (the pseudo-descriptors) — restore guest state.
- `es.all` and `ds.all`, if the host was in 32-bit mode (`efer.lma==0`) when it issued the SVM instruction. Again, save the hidden state of the segment registers.
- `dr7` — restore the guest's breakpoint state.
- `v_tpr`, the guest's virtual TPR.

The processor checks the loaded state for consistency; if an illegal mode is detected[1], the processor exits guest mode again — without saving the guest state, and storing an error indication in the VMCB reason code. On vm-exit, the above guest state is stored back into the VMCB, with `ds` and `es` (including their hidden state) only stored if the host mode (which we're about to switch to) is in 32-bit mode (`efer.lma==0`).

4.1.3 Control Bits

In addition, the following state is loaded from the VMCB in order to further control execution of the guest:

- the maximum number of instructions the guest is permitted to execute. IMPLEMENTATION DETAIL: it is unlikely that this feature will be supported in the first revision of VM support.
- 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). IMPLEMENTATION DETAIL: it is not clear whether this feature will be supported in the first revision of VM support.
- `v_irq`, `v_intr_prio`, `v_intr_vector` — fields that indicate, respectively, whether a virtual INTR (maskable) interrupt is pending for the guest, and if so, what its priority and vector number are. (Note that the related field, `v_tpr`, is both loaded and written back by SVM, since unlike these fields here, it is state that the guest can actually change.)
- the TLB ASID (address space ID) to use while running the guest. Initially, we will support only a single ASID bit for the use of VM, to distinguish host and guest space. We plan to add more guest ASIDs later. An ASID of "0" indicates host ASIDs. (Note: there will be a mechanism, e.g. CPUID level, that allows the VMM to determine how many ASIDs are implemented.)
- a flag indicating whether to flush the TLB of all guest entries before running the guest.
- the intercept vector (which may also involve loading pointers to tables protecting I/O ports and MSR accesses).
- SVM unconditionally sets GIF to "1". (Conversely, when exiting from a guest back to the

1. If the context in which SVM loads cr3 has PAE enabled, the loading of cr3 will be accompanied by loading the four toplevel PDPEs; if any of those have reserved bits set, the guest will also exit back to the host.

VMM, the microcode unconditionally sets GIF to "0", to enable the VMM to atomically complete its world-switch.)

4.1.4 SVM Intercepts

SVM loads the intercept vector from the VMCB; this vector defines what events in the guest cause an exit to the VMM. (On exit from the guest, microcode clears the internal intercept registers, so no host operations will be intercepted.)

This section gives a tentative list of intercepts. Note the fine distinction between SVM intercepting an operation in the guest that can also cause an exception (which might have higher priority than the SVM intercept on the operation) versus SVM intercepting that very same exception.

IMPLEMENTATION DETAIL: we may coalesce several of the intercepts so they're controlled by a single bit in the VMCB. Details on individual intercepts are still TBD, in Section 4.2.

The following events inside the guest are always intercepted

- `vmmcall` instruction (the equivalent of SEM's `smcall` instruction)

The intercept vector allows the VMM to optionally intercept the following operations:

- halt
- shutdown
- any control register (CR) or debug register (DR) read or write
- selective CR0 write (allow changes to CR0.TS, intercept other changes)
- any exception (vectors 0–31)
- any external interrupt (INTR, NMI, INIT, SMI); in the case of INTR (maskable interrupts), only if the interrupt is enabled as defined in Figure 3 on page 39. The intercept on INTR also controls whether virtual interrupts are enabled, and whether writes to `cr8` write `v_tpr` instead of the physical TPR in the localAPIC.
- invalpg
- rdtsc
- rdpmc
- rsm
- iret
- any use of FP except fwait
- fwait
- ferr_freeze
- pause
- cpuid

- read or write of idtr, gdtr, ldtr
- invd
- skinit
- stgi, clgi
- any task switch
- IOIO: check-enable bit, plus pointer to I/O permission table
- MSRs: check-enable bit, plus pointer to MSR permission table

We can expect that some of these bits will be set in any "reasonable" VMM; however, giving the user explicit control makes SVM a useful instruction for verification, since it will simplify getting the processor into a well-defined state.

4.1.5 Consistency Checks

SVM and VMLOAD need to perform consistency checks on host and guest state, very much like RSM performs checks on the new state. Illegal states can cause either of an exit to the host (with errorcode), a #GP fault, or (in the worst case) a processor shutdown.

To simplify emulation of real mode, we allow SVM (and SVM only) to load a guest (and guest only) value of `cr0` with PE==0 but PG==1, otherwise an illegal setting.

4.2 Detailed Intercept Operation

This section defines the detailed handling of the defined SVM intercepts, in particular where in the microcode flow the SVM intercept check is done (relative to other operations and exception checks). All intercepts, when triggered, write an "exitcode" into the VMCB (for most intercepts, this could just be the index into the intercept vector). If an intercept provides additional information, in either MSRs or the VMCB, it is noted here.

IMPORTANT: when an intercept triggers on a given (guest) instruction, the processor state saved in the VMCB must be the state as of before the instruction in question. Specifically, the saved eip must point at the intercepted instruction.

4.2.1 Intercept Priority — Design Guideline

A critical decision for the placement of intercept checks is where they fall with regards to existing exception checks for a given instruction. Some of these exception checks are used by existing x86 (partial) virtual machine like environments, e.g., v86 mode and SMM.

As a general rule, we should place SVM intercepts at the lowest priority that still allows us to catch all the instances of the instruction or event that we are interested in. If we give the intercept too high a priority, the VMM software will have to check any lower-priority exceptions, and possibly reflect the event "back up" into the guest. On the other hand, if we give the SVM intercept too low a priority, the VMM might not have enough capabilities to function.

IMPLEMENTATION DETAIL: by and large, SVM intercepts will likely be checked after "simple" exceptions (such as requiring CPL==0) have been checked, but before memory-based exceptions (such as page faults), or (in some cases) exceptions based on specific operand values. One important exception from this guideline is SMI interception, see Section 4.10.

The remainder of this section details where in the microcode flow the individual SVM intercepts are checked (if enabled, that is).

4.2.2 vmmcall, halt, shutdown

Any exceptions are checked first, then the SVM intercept.

4.2.3 mov to/from cr0, lmsw, smsw, clts

Check exceptions first (CPL, illegal bit combinations, etc.). For lmsw and smsw, check SVM intercepts before checking memory exceptions.

4.2.4 Selective cr0 write intercepts

The selective write itercept on cr0 triggers only if a bit *other* than `cr0.ts` is being changed by the write. In particular, this means that `clts` does not check this intercept. IMPLEMENTATION DETAIL: the `lmsw` instruction may always intercept cr0 writes, regardless of the value.

4.2.5 mov to/from cr3 (including task switch)

Check most exceptions first, then the SVM intercept. If the SVM 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 SVM intercept, so the VMM needs to be prepated to check that condition itself.

Note that loading cr3 via SVM, or on exit from SVM, is *not* subject to the intercept check!

4.2.6 mov to/from other cr

All normal exception checks should take precedence over the SVM intercepts.

4.2.7 mov to/from debug register

All normal exception checks should take precedence over the SVM intercepts.

4.2.8 External Interrupts (INTR, NMI, SMI, INIT)

External interrupts, when being intercepted, cause an exit from the guest; the external interrupt signal is held pending so that the interrupt can (eventually) be taken in the VMM. See Section 4.10 for details on SMI handling, Section 4.9.3 for virtualization of maskable interrupts (INTR), and Section 4.9.5 for the INIT-redirection feature; also Section 4.5 for details on the global interrupt flag.

For an intercepted SMI, the SVM exitcode should distinguish whether the SMI was caused internally (by I/O Trapping), or asserted externally.

Note that the INTR intercept also controls the virtual interrupt facility, and associated (hardware) virtualization of the localAPIC's TPR register.

4.2.9 Exception Intercepts

For intercepted exceptions, the hardware delivers two pieces of information in the VMCB besides the exitcode: the vector number of the exception (in the VMCB's `exitinfo[0]` field), and, where defined, the errorcode that would have been pushed onto the exception stack (in `exitinfo[1]`). The values of `rflags`, `cs.sel` and `rip` saved in the VMCB on an exception-intercept should be those that would otherwise have been pushed onto the exception stack frame.

Typically, the intercept should occur *before* the exception modifies any guest state, e.g., pushes an exception stack frame. However, some exceptions write additional information into special registers (e.g., #PF writes cr2, #MC writes various error reporting registers, etc.); in some cases the intercept check should occur *after* those registers should be written. (Examples: in the case of #PF, to speed up shadow page table handling; in the case of #MC to provide information that the VMM cannot otherwise regenerate by emulating a faulting instruction.)

Note that only exceptions 0 through 31 can be intercepted.

4.2.9.1 #DE (Divide By Zero)

No special registers are written. The errorcode is undefined.

4.2.9.2 #DB (Debug)

IMPLEMENTATION DETAIL: Due to microcode constraints, support for this intercept may be limited. The easiest implementation would likely be one where an intercept replaces the conventional fault or trap delivery, *after* `dr6` has been written.

4.2.9.3 NMI (External NMI Signal)

NMIs are already dealt with by the interception of external interrupts; intercepting them at the exception-vector level would be redundant.

4.2.9.4 #BP (Breakpoint)

No special registers are written. The errorcode is undefined.

4.2.9.5 #OF (Overflow)

No special registers are written. The errorcode is undefined.

4.2.9.6 #BR (Bound-Range)

No special registers are written. The errorcode is undefined.

4.2.9.7 #UD (Invalid Opcode)

No special registers are written. The errorcode is undefined.

4.2.9.8 #NM (Device-Not-Available)

No special registers are written. The errorcode is undefined.

4.2.9.9 #DF (Double Fault)

No special registers are written. The errorcode is undefined. Note that the `rip` value saved in the VMCB is undefined (as is the case for the `rip` value pushed on the stack for #DF exceptions).

4.2.9.10 Vector 9 (Reserved)

No used, so no need to intercept.

4.2.9.11 #TS (Invalid TSS)

No special registers are written. The errorcode is undefined. Note that the `rip` value saved in the VMCB may point to either the instruction causing the task switch, or the first instruction of the incoming task.

4.2.9.12 #NP (Segment Not Present)

No special registers are written. The errorcode (saved in `exitinfo[1]`) is the segment selector index, as would be pushed on the stack by a #NP exception.

4.2.9.13 #SS (Stack Fault)

No special registers are written. The errorcode is the same as would be pushed on the stack by a #SS exception.

4.2.9.14 #GP (General Protection)

No special registers are written. The errorcode is the same as would be pushed on the stack by a #SS exception.

4.2.9.15 #PF (Page Fault)

The SVM intercept should be tested *after* `cr2` is written, but *before* other guest state is modified. The errorcode (saved in `exitinfo[1]`) is the same as would be pushed onto the stack by a non-intercepted #PF exception.

4.2.9.16 #MF (X87 Floating Point)

This intercept should trigger after the floating point status word is written as in normal execution. The errorcode is undefined.

4.2.9.17 #AC (Alignment Check)

No special registers are written. The errorcode is undefined.

4.2.9.18 #MC (Machine Check)

The machine-check intercept should be tested after the various MCA registers have been written, but before other guest state is modified. A machine-check incurred while running a guest should exit to the VMM wherever possible, and only shutdown the processor where this is not a reasonable option.

IMPLEMENTATION DETAIL: details (on what state is written before the intercept, and when shutdowns occur even inside a guest) are likely to be guided by implementation concerns.

4.2.9.19 #XF (SIMD Floating Point)

This intercept should trigger after the SIMD point status word (`mxcsr`) is written as in normal execution. The errorcode is undefined.

4.2.10 invalpg

Check all exceptions (#GP and #UD) before the SVM intercept check.

4.2.11 rdtsc, rdpmc

Check all exceptions before the SVM intercept check.

4.2.12 rsm

Check the SVM intercept at the start of the microcode flow, i.e., even *before* checking exceptions. The VMM is expected to emulate the entire `rsm` instruction, complete with exception checks. The reason is that the VMM may need to intercept `rsm` in situations where the processor is not physically in SMM mode (i.e., where `rsm` is an undefined instruction, and checking for exceptions before the SVM intercept would trigger a #GP in the guest, as opposed to intercepting).

4.2.13 iret

For `iret`, the SVM intercept is checked at the beginning of the microcode flow, i.e., before any exceptions have been checked.

4.2.14 FP Intercept

Act as though `cr0.ts` was set, but instead of generating an exception in the guest, exit to the VMM — before any guest state is modified.

4.2.15 Ferr_Freeze

Intercept when the processor freezes due to assertion of FERR (while IGNNE is deasserted), i.e., while the processor is waiting for an external interrupt to be "unfrozen".

4.2.16 cpuid

Intercept check can come first; there are no exceptions to check.

4.2.17 pause

IMPLEMENTATION DETAIL: this instruction may or may not be interceptable in initial implementations of SVM support.

4.2.18 read or write of idtr, gdtr, ldtr

Check SVM intercept only after having checked most normal exceptions.

4.2.19 skinit

Since this is likely not a frequent operation, it should suffice to check the SVM intercept at the beginning of the microcode flow; the VMM can be expected to check exceptions in software.

4.2.20 clgi, stgi

Check exceptions first (the instructions are only available at cpl-0), then check SVM intercept.

4.2.21 Task Switch

Due to their complexity, task switches can trigger several different intercepts (cr0 write, cr3 write, ldtr write, memory accesses). Task switches are not generally restartable, a condition that is highly desirable for VMMs. We have several choices:

- we can rewrite the task switch code so as to be restartable (for any conceivable intercept) when in guest mode.
- we can always intercept all task switches before any guest state has changed, and have the VMM emulate the switch in software (task switches should be rare enough in modern OSs). IMPLEMENTATION DETAIL: this is the preferred implementation.
- the VMM may decide that it has enough information about, or cooperation from, the guest that it can "pick up" a task switch from an intercept "in the middle". We should not rely on this mode of operation for correctness, but provide an option to support it.

Note that there are several places (INTx, JMP, CALL, exceptions, interrupts) where a task switch can occur.

4.2.22 invd

This instruction can cause loss of data from the caches, so the VMM should typically intercept it. Exceptions are checked before the SVM intercept check.

4.2.23 IOIO Intercepts

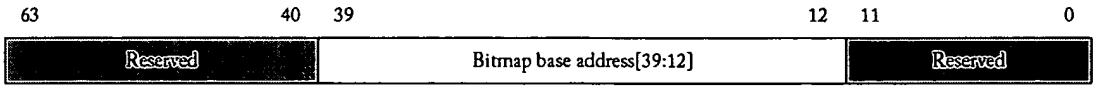
The VMM can intercept IOIO instructions (IN, OUT, INS, OUTS) on a port-by-port bases via the *SVM I/O permissions map*. This is essentially the same feature as was available in SEM. If the `ioio_prot` bit is set in the VMCB's intercept vector, the SVM instruction must also load the `ioio_table` field from the VMCB into the (read-only) VM_IOPM_BASE MSR.

4.2.23.1 I/O Permissions Map

The IO Permissions Map occupies 8K bytes of contiguous physical memory. The table is structured as a linear array of 64K bits (two 4KB pages) and must be aligned on a 4KB boundary. 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. The *physical* base address of this table is contained in the VM_IOPM_BASE MSR; the MSR is written by the SVM instruction.

Table 1: VM_IOPM_BASE MSR.



4.2.23.2 IN and OUT Behavior

Any IOIO operation checks for exceptions and SVM intercepts in the following order

- check v86, traditional exceptions, IOPL, TSS-bitmap, etc. (definitely should come first)
- the I/O restart information should be written *before* the SVM intercept checks
- if the `ioio_prot` intercept is set, check whether the VM_IOPM_BASE table permits access to the given port. If this intercept triggers, the IOIO operation must *not* be marked as having been interrupted by an SMI.
- for INS and OUTS, check memory permissions (could be done @ higher priority)
- check for internal synchronous SMI, i.e., I/O Trapping (should come after SVM checks)
- check for external SMI, synchronous or asynchronous (could be done @ higher priority)
- debug breakpoints should be checked *after* the SVM intercept checks.

4.2.24 MSR Intercepts

The VMM can intercept RDMSR and WRMSR instruction via the *SVM MSR permissions map.* This is essentially the same feature as available in SEM, except that the table contains two bits per MSR, one to intercept MSR reads, and one to intercept MSR writes. If the `msr_prot` bit is set in the VMCB's intercept vector, the SVM instruction must also the `msr_table` field from the VMCB into the (read-only) VM_MSRPM_BASE MSR.

4.2.24.1 MSR Permissions Map

Note: in a change from SEM, the MSR map uses two *bits per MSR, one (the low-order one) to control read accesses, and one (the high-order one) to control write accesses.*

A simple flat bitmap will not suffice for the MSR permissions map — the MSR address space is 32 bits (allowing for 4G MSRs). AMD processors use 5 widely-separated ranges within that space:

Table 2: AMD Processor MSR Addresses

0000_0000 - 0000_0400	Pentium-compatible MSRs
C000_0080 - C000_0102	K6 MSRs (and SYSCALL)

Table 2: AMD Processor MSR Addresses

C001_0000 - C001_0113	K7/K8 public MSRs
C001_1000 - C001_1027	K7/K8 private MSRs (password protected)
C001_2000 - C001_2200	K7/K8 hidden MSRs (password protected)

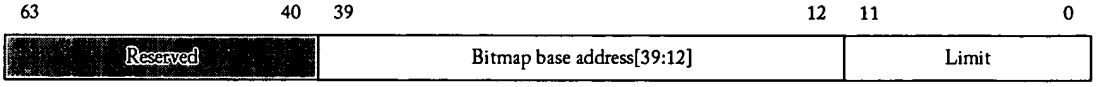
Note that a relatively small number of MSRs are implemented in each range. Therefore the MSR bitmap consists a number of smaller separate bitmaps of 2K bytes each covering a defined range of 8K MSRs. Four of these smaller bitmaps can reside in two physical pages (8KB, covering 32K MSRs). One 8K range is used for the Pentium compatible MSRs, the next for the K6 MSRs. The K7/K8 public and private MSRs are covered by the third range. The K7/K8 hidden MSRs are defined to be always protected when MSR interception is active, for both read and write access, therefore they do not need a bitmap. In each range, attempts to read or write any MSR beyond the range of the bit map (8K entries) will automatically cause an intercept (if the `msr_prot` intercept is active). The ranges of the MSR bitmap are defined below:

Table 3: Ranges of MSR Permissions Map

Byte offset	MSR range	Current usage
000-7ff	0000_0000 to 0000_1FFF	Pentium-compatible MSRs
800-fff	C000_0000 to C000_1FFF	K6 MSRs (and SYSCALL)
1000-17ff	C001_0000 to C001_1FFF	K7/K8 public/private MSRs
1800-1fff	XXXX_XXXX to XXXX_XXXX	future K9 and K10 MSRs

Currently, all MSRs can be mapped in two pages of physical memory with room left over for expansion. If necessary for future MSR expansion, additional pages can be added to the bit map. Any additional pages shall be contiguous with the first page, so that a single MSR can be used to point to the base of the MSR permissions map. The VM_MSRPM_BASE MSR contains the *physical* address of the bit map; the MSR is written by the SVM instruction.

Table 4: VM_MSRPM_BASE MSR.



Future MSR ranges are supported by using expansion room in the first page of the bit map or adding additional pages to the bit map. It is not necessary to pre-define what those ranges might be or how many ranges exist. The limit field in bits 11:0 of VM_MSRPM_BASE specifies how many *pages* the VMM has defined for the bitmap. Any MSRs that would map to protection bits outside the defined pages are treated as protected (accessible only if the `msr_prot` intercept is inactive, typically in VMM mode). If the limit field is zero, then no bitmap is defined, no bitmap lookup is performed, and all MSRs are protected (if the `msr_prot` intercept is active, that is).

If the `msr_prot` intercept is *inactive*, all MSRs are fully accessible.

4.2.24.2 RDMSR and WRMSR Behavior

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, etc.)

The SVM intercepts could be done at lower priority, however, the arrangement proposed here may simplify microcode implementation.

4.3 VMSAVE and VMLOAD Instructions

These two instructions also take the address of a VMCB in the (implicit) `eax` operand. The instructions are intended to complement the state save/restore abilities of the SVM instruction. In particular, they must provide access to all hidden state that s/w cannot otherwise access (and thus save itself). As a performance optimization, VMSAVE and VMLOAD may also handle s/w accessible state (but they might do it faster in microcode than x86 code could).

VMSAVE saves the following (presumed guest) state to the VMCB pointed to by eax:

- `es`, `ds` (including all hidden state), but only if the host is in long mode; if the host is in 32-bit mode, the SVM instruction will already have saved the guest `ds` and `es` (and overwritten both registers with host state)
- `fs`, `gs`, `tr`, `ldtr` (including all hidden state)

VMLOAD loads the corresponding state from the VMCB (`es` and `ds` are only loaded if the host is in long mode).

IMPLEMENTATION DETAIL: we may add a bitmap to the VMCB so s/w can control what additional state is handled by VMSAVE/VMLOAD: some state is saved and restored more quickly in microcode than x86 code.

4.4 TLB Control

The TLB will be tagged with at least one ASID bit to distinguish host and guest address spaces. Normal TLB invalidations (write to `cr3`, write to `cr4`, `invalpg`, etc.) should only affect translations for the current ASID (the current ASID after reset is "0"). In particular, this means that the VMM is responsible for explicitly invalidating any guest translations (translations for a ASIDs other than the

current one) that may be affected by a change in host translations; there are two mechanisms available.

4.4.1 Guest TLB Flush

The VMM can request that *all* guest translations be flushed from the TLB by setting the appropriate control bit in the VMCB before executing a SVM instruction. **IMPORTANT**: this must flush *all* guest translations, even if they are marked "global"!

Note that a write to `cr3` by the guest only needs to flush that guest's translations; it is recommended that implementations not flush any other ASID's translations.

4.4.2 Selective TLB Invalidation

The VMM needs to be able to selectively invalidate the (guest) TLB mapping for a given virtual page; i.e., hardware needs to provide a mechanism to invalidate a TLB mapping for a given virtual address and given ASID.

IMPLEMENTATION DETAIL: to minimize impact on existing designs, some implementations may invalidate more than just a single page, or one page in each of multiple ASIDs.

4.5 Global Interrupt Flag, STGI and CLGI Instructions

These instructions set and clear, respectively, the global interrupt flag, GIF. This is a carry-over from SEM, but note that the instructions are now available in plain CPL-0 mode! Table 5 shows how the setting of GIF affects the handling of interrupts and exceptions.

Table 5: Effect of 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 6 on page 41
NMI	held pending until gif==1	normal operation, see Table 7 on page 41
External SMI	held pending until gif==1	normal operation, see Table 8 on page 42
Internal SMI (I/O Trapping)	ignored and discarded	normal operation, see Table 8 on page 42
INTR	held pending until gif==1	normal operation, see Figure 3 on page 39
Machine Check	shutdown	if intercept_MC[1]: vm-exit else: normal operation.
DBREQ# (enter HDT)	normal operation[2]	normal operation
A20M	normal operation[3]	normal operation
Other implementation-specific but non-architecturally-visible interrupts (STPCLK, IGNNE toggle, ECC scrub)	normal operation	normal operation

1. Either explicit intercept_MC bit, or intercept for exception #18 if we implement per-vector exception intercepts.
2. VMCR.DPD always controls DBREQ_
3. VMCR.DIS_A20M controls A20 masking

4.6 SKINIT Instruction

This instruction is carried over from SEM essentially unchanged; its operation is described in detail in Section 4.13.5.

4.7 VMMCALL Instruction

This instruction always causes an intercept to the VMM (i.e., save current state and reload host state) — it is meant as a way for a guest to explicitly call the VMM. (In the SEM spec, this instruction was called "smcall".)

4.8 New Processor Mode: Paged Real Mode

To facilitate virtualization of real mode, we allow SVM to load a guest `cr0` value with PE==0 but PG==1. This processor mode shall behave in every way like real mode, with the exception that paging is applied (as opposed to inserting 1:1 translations in the TLB, which is what happens in "normal" real mode). The intent is that the VMM run the guest at CPL0, and with all page faults intercepted. The VMM is responsible for setting up a shadow page table that makes guest physical memory appear at the proper virtual addresses inside the guest.

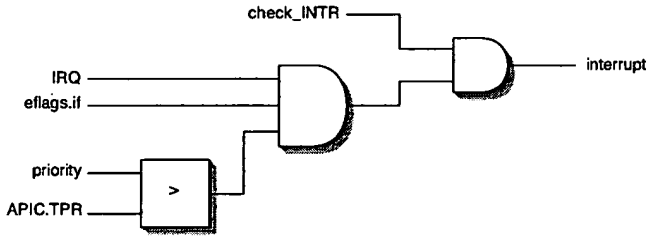
IMPORTANT: all the SVM intercepts must still work in paged real mode (in case operations have separate microcode flows for real and protected mode, intercept checks must be placed in both flows).

IMPORTANT: the DEV table (Section 4.12.2) is *not* consulted by the CPU, even in real mode. This is different from what was done in SEM. The VMM is responsible for never letting the guest enter (physical) real mode; paged real mode must be used instead.

4.9 Interrupt and localAPIC Support

Figure 2 captures the current interrupt logic: an interrupt is asserted to the core if an external interrupt request is pending ("IRQ"), interrupts are enabled in eflags.if, the priority of the interrupts is greater than the TPR threshold in the APIC, and interrupts are being checked at the current instruction boundary ("check_INTR"); the latter signal is used to implement the one-instruction interrupt lockout window after `sti` and `mov-to-ss` instructions.

Figure 2. Current masked interrupt (INTR) logic.



To support efficient virtualization of interrupts, we need to make the following changes to the processor core and the local APIC.

4.9.1 Physical (INTR) Interrupt Masking in eflags

To prevent the guest from blocking maskable interrupts (INTR) via the `eflags.if` bit, we propose the following:

- the masking of (physical) INTR interrupts shall be performed by a new bit, PHYS_IF
- if INTR interrupts are *not* being intercepted, writes to `eflags.if` shall also write PHYS_IF to

the same value

- in INTR interrupts are being intercepted, writes to `eflags.if` shall leave PHYS_IF unchanged

Finally, physical interrupts received while a INTRs are being intercepted shall cause the guest to exit (with exit code VMEXIT_INTR), so the VMM can process the interrupt.

4.9.2 Virtualizing APIC.TPR

We introduce a new register, `v_tpr`, that is accessed in lieu of the localAPIC's TPR when the processor does not currently own INTRs (as would typically be the case in guests). In long mode, this is easily accomplished, since the TPR is mapped to control register `cr8` — so we only need to change the microcode for accessing `cr8`.

However, in 32-bit mode, the TPR is a memory-mapped register. Typically, we would virtualize memory-mapped devices by not mapping their addresses in the guest. A guest access to that region would then cause a #PF intercept to the VMM, which would walk the guest page tables to determine the physical address, recognize the physical address as belonging to the APIC, and finally invoking software emulation code. Not only is this slow, but it also would trap all accesses to the APIC page — yet we only want to trap accesses to APIC registers *other* than the TPR.

IMPLEMENTATION DETAIL: it is highly unlikely that we will hardware support for virtualizing memory-mapped accesses to the localAPIC's TPR; a VMM would have to intercept all accesses to the APIC page, and emulate the TPR access in software.

A possible solution is to make `cr8` available to 32-bit code. To achieve best performance, guests would need to be modified to use `cr8` instead of the memory-mapped TPR.

4.9.3 Virtual (INTR) Interrupt Support

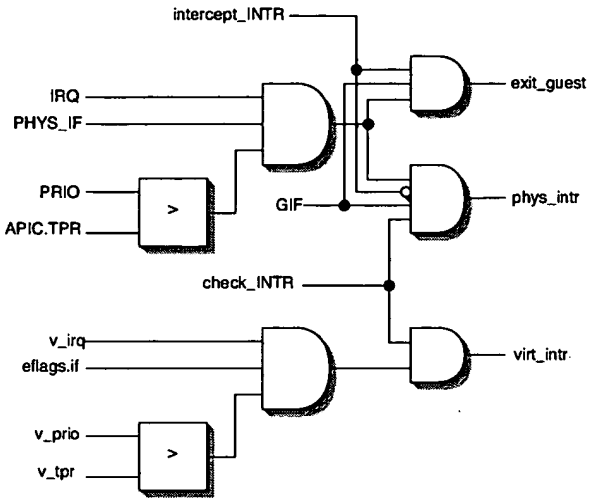
We need efficient support by which the VMM can inject a (virtual) interrupt into a guest; this includes handling guest writes to its (virtual) localAPIC's TPR (for raising and lowering the threshold on interrupt priority). We propose the following:

- if the INTR intercept bit is zero, the processor is said to "own" the maskable interrupts, and can take them directly (as well as directly manipulate the localAPIC). If the INTR intercept is "1", physical interrupts cause an exit to the VMM, the virtual interrupt mechanism described in this section is enabled, and references to `cr8` access `v_tpr` instead of the physical TPR in the local-APIC. This bit is always "0" in host mode (vmexit resets it to "0", like all intercepts).
- three fields in the VMCB indicate whether there is a virtual interrupt pending, and if so, what its vector# and priority are. The SVM instruction loads this information into on-chip registers, `v_irq`, `v_intr_prio`, and `v_intr_vector`.
- if the INTR intercept is enabled, `v_irq` and `v_intr_prio` indicate that there is a virtual

interrupt pending, and its priority is greater than the value in `v_tpr`, the processor shall take a INTR interrupt, with the sole difference from "normal" interrupt handling that the interrupt vector will be obtained from the `v_intr_vector` register (as opposed to running an INTAK cycle to the localAPIC).

Figure 3 below shows the revised logic for handling maskable interrupts. When physical INTRs are being intercepted, what used to cause a physical interrupt now merely causes an exit from the guest. Physical interrupts can only occur when no longer in guest mode (and allowed by GIF, the global interrupt flag); physical interrupts are masked by the new `phys_if` bit, and `eflags.if` is relegated to masking virtual interrupts only. Note that virtual interrupts are not affected by GIF. (Also, virtual interrupts can only happen when a guest is running: microcode always clears `v_irq` is when exiting the guest.)

Figure 3. Logic for handling physical and virtual masked (INTR) interrupts.



An additional change is that in response to a `phys_intr`, the processor runs an INTAK cycle to the local APIC in order to obtain the interrupts vector; for a `virt_intr`, the processor will read the interrupt vector from the `v_intr_vector` register. After that, the processing of virtual and physical interrupts is identical.

Note: the exit_guest signal must of course only be recognized on instruction boundaries.

Note: in the above logic, a guest can hold off a physical interrupt for one cycle by executing an `sti` or `mov-to-ss` instruction. However, the (microcode) logic driving the `check_INTR` signal is designed such that a back-to-back sequence of either of these instructions can only suppress interrupt checking for one (or a few cycles), as opposed to forever. Hence, it is safe to *not* include two copies of this signal (one each for host and guest).

4.9.4 Interrupt Masking in localAPIC

Once guests have direct access to devices, interrupts will arrive at the localAPIC that can only be dismissed by the guest that owns the device causing the interrupt. To prevent malicious or buggy guests from blocking other guests' interrupts, the VMM must be able to mask pending interrupts in the localAPIC, so they do not participate in the prioritization of (other) interrupts.[1] We propose the following

- add a 256-bit IM (interrupt mask) register to the localAPIC; this register resets to all-ones; s/w can read and write the IM
- when computing the PPR (APIC processor priority), only examine ISR bits that are not masked off in the IM (i.e., use "ISR & IM" instead of ISR)
- when determining the highest-priority pending interrupt to deliver to the processor, only examine IRR bits that are not masked off in the IM (i.e., use "IRR & IM" instead of IRR)
- when determining the vector to send to the processor in reply to an INTAK cycle, only examine IRR bits that are not masked off in the IM (i.e., use "IRR & IM" instead of IRR)
- change the localAPIC logic that prioritizes interrupts to be delivered to the processor, such that effectively all bits in the ISR are first ANDed with their corresponding bit in the IM.
- allow the CPU to issue "specific" (to use a legacy PIC term) EOIs to the localAPIC, so it can clear pending interrupts in any order, as opposed to always targeting the highest-priority one.

4.9.5 INIT Support

The INIT signal is similar to SMI# in that it interrupts the processor after completion of the current instruction and causes an unconditional control transfer. On non-VM x86 processors, INIT reinitializes the Control Registers, Segment registers and GP registers 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.

To maintain the security of such pages, the VMM can request (by setting the r_init bit in the VM_CR MSR) that INITs be "redirected" and turned into #SX interrupts. The #SX interrupt is not masked by EFLAGS.IF, but it is masked by the Global Interrupt Flag. The intent of this is to allow the VMM to gain control whenever an INIT has been requested by hardware or software. After taking any such steps, the VMM must 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

1. Note that the VMM cannot simply send an EOI immediately, before the guest has serviced the interrupt — the device would still be asserting the interrupt.

by the INIT pin may also be initiated by an incoming APIC INIT interrupt; the changes described here apply in either case. Table 6 below summarizes the handling of INITs.

Table 6: INIT Handling in Different Operating Modes

GIF	intercept INIT?	INIT redirect?	processor response to INIT
0	x	x	hold pending until GIF = 1
1	1	x	hold pending, exit guest, code VMEXIT_INIT
	0	0	taken normally
		1	turn into #SX exception, clear INIT-pending bit.

4.9.6 NMI Support

NMI support is simpler than INIT, since it already shows up as an exception, so the VMM can regain control via the NMI exception handler; no "redirect" mechanism is needed. The only modification is that when intercepted (as would typically be the case when running a guest), NMIs cause an exit from the guest, but are held pending.

Table 7: NMI Handling in Different Operating Modes

GIF	intercept NMI?	processor response to NMI
0	x	hold pending until GIF=1
1	1	hold pending, exit guest, code VMEXIT_NMI
	0	taken normally

4.10 SMM Support

Support for virtualization SMM is complicated by several factors.

- to support proper containerization of SMM code, the VMM needs to gain control upon entering and exiting (physical or virtual) SMM, so it can set up "paged real mode" and associated page tables (see Section 4.8).
- the hypervisor may not trust the SMM code that gets executed in response to either a guest or host originated SMI. Therefore, we need a mechanism that intercepts even SMI events *within* the hypervisor itself.
- the hypervisor needs to be able to access SMRAM space, and must keep its code and data structures out of physical pages that can be "switched" to a different area while in SMM mode. Note that this is somewhat platform-specific information, and must be provided externally to the

hypervisor from a trusted source (e.g., the Secure Loader).

4.10.1 Sources of SMI

Various events can cause an assertion of SMI; we classify them into three categories

- internal, synchronous (a.k.a. I/O Trapping) — 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 via MSRs, and as such can be brought under the control of the VMM.
- 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.

Microcode for IOIO operations saves relevant processor state (eip, ecx, esi, edi) as of the start of each IOIO operation, as well as a flag indicating whether an IOIO operation was aborted by an SMI. SMM microcode already uses this information to save I/O Restart information in the SMM save area. We propose making this information visible via MSRs, for use by a VMM.

4.10.2 Response to SMI

How hardware responds to SMIs is a function of whether the processor is in guest mode and whether interrupts are enabled globally. Table 8 below shows a summary of all modes.

Table 8: 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_INT	hold pending, exit guest, code VMEXIT_SMI_EXT
	0	taken normally	taken normally

There are three possible VMM software actions (though not all are legal for all cases)

- reflect a virtual SMM back into the guest (probably only in case 1a)
- allow direct control transfer to the platform SMM code
- allow control transfer to a SMM trampoline set up by the VMM, which reflects the SMI to the platform SMM code, running inside a container.

4.10.3 Virtual SMM Mode in a Guest

The only time a guest operation can *cause* an external SMI is if that guest is in fact allowed to access a physical device directly. In that case, the correct response is probably to run the platform SMM code, either directly (if trusted) or inside a container. However, there are cases where we may want to run *guest* SMM code. This will typically not be in response to an external SMI, but a result of the guest having set up I/O trapping, e.g., in order to provide a virtual device itself.[1]

Placing a guest into virtual SMM is a matter of

- setting up shadow page tables so as to mimic SMRAM for the guest
- copying guest state from the VMCB into the guest's (virtualized) SMM save area
- running the guest in "paged real mode" until an `rsm` occurs
- need to also intercept `iret` inside guest, which reenables (virtual) NMIs for the guest

Note that at no time does the processor actually enter physical SMM; the SMRAM remapping is all virtualized via the shadow page tables.

4.10.4 Containerizing Platform SMM

When the VMM does not trust the platform SMM code, it can still invoke the platform SMM code, either for its own benefit, or on behalf of a guest. It does so by running the SMM code in a container (i.e., in guest mode).

Software architecture note: if software intends for one container to handle SMM events that occurred inside another container, this probably places restrictions on how the SMM code must be written. Specifically, there is an issue as to what environment "outside" of the SMM areas the SMM code should see — that of its "own" container, or (parts of) the container in which the event was originally detected. Since we run containerized SMM in "Paged Real Mode" (see Section 4.8), the hypervisor can freely remap the address space visible to the contained SMM.

At this point, we propose two different options for SMM support.

4.10.4.1 Minimal Support

For the absolute minimum support, we rely on the VMM to create its own trusted SMM handler, and use that as a trampoline to invoke the platform SMM code inside a container (similar to virtualizing SMM in a guest, Section 4.10.3). The h/w action of physically entering SMM takes care of providing all the information about pending I/O (the SMM microcode would save all the

1. Instead of letting the guest access the MSRs that set up I/O Trapping, the hypervisor can intercept the guest's attempts to set up I/O trapping, and set up an SVM IOIO intercept instead. In that case, the hypervisor could reserve the I/O-trapping MSRs for its exclusive use (no save/restore needed for guests!), it does need to reflect IOIO intercepts back to the guest, as a virtual SMIs.

information in the VMM's SMsave area), so we would not absolutely *need* new MSRs for accessing that information. No new SVM features are required, but SVM must *not* alter the state of SMM when switching between host and guest; it is the responsibility of the VMM to ensure that none of its code or data "disappears" when the SMRAM areas are mapped or unmapped.

WARNING: this requires that the VMM always be able to reassign SMM_BASE; this will render useless the feature for "locking" SMRAM. In processors with the proposed VM support, SMM code should not rely on SMRAM locking for security (e.g., for hiding secrets).

Note that with this level of h/w support, we can not enter platform SMM code in response to an internal synchronous SMI occurring inside a guest, since that event does not cause the processor to enter SMM (even after exiting to the host and setting GIF:=1). This deficiency could be fixed if the VMM had a means for programmatically triggering an SMI (e.g., by writing an MSR).[1]

Another drawback of this proposal is that entering and SMM would take about three times as long as it does without virtualization support (see outline of s/w actions in Section 5.10.3).

4.10.4.2 Advanced Support

The following h/w support could speed up handling of internal SMIs by 30% (so they're only about twice as expensive as SMIs without virtualization); this would also simplify the VMM code since it would allow the VMM to reserve the I/O Trapping MSRs for its own exclusive use, which in turn means that the VMM would not have to save/restore these registers when switching guests (another performance gain). Finally, all internal I/O traps could be virtualized using a single mechanism, namely the SVM I/O permissions map.

We propose the following hardware mechanisms to give the hypervisor full control over SMM.

- a mechanism (MSR?) that allows s/w to independently control
  - 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)
  - clearing the SMI-pending flag in the processor.
- MSRs that give s/w access to state saved by the most recent IOIO instruction; enough to restart that instruction.
- no new SVM features are required, but SVM must *not* alter the state of SMM when switching between host and guest; it is the responsibility of the VMM to ensure that none of its code or data "disappears" when the SMRAM areas are mapped or unmapped.

1. As a hack, the VMM could trigger an SMI by setting up I/O Trapping at a known (but harmless) I/O port, then accessing that port. However, this action would destroy the information about any aborted I/O operation inside the guest, thus complicating the emulation thereof.

WARNING: these h/w mechanisms will render useless the feature for "locking" SMRAM, as they give CPL-0 code the ability to inspect the SMRAM areas at will. In processors with the proposed VM support, SMM code should not rely on SMRAM locking for security (e.g., for hiding secrets).

4.10.5 SMM Software Architecture Notes

It is important to note that with the h/w controls over (physical) SMM, as proposed in Section 4.10.4, invoking *physical* SMM inside a container is rather similar to completely virtualizing SMM: in the latter, the mapping and unmapping of SMRAM can (largely) be simulated via the shadow page table, and in either case, we simulate real mode via appropriate intercepts to the hypervisor.

There is an opportunity here to clean up the existing SMM architecture: the hypervisor can define a fairly general interface that allows one container to intercept events in another container; the actual details of execution mode in either container can be decoupled, and it would be easy to define an "SMM-like" environment that comes up, say, in protected mode with paging enabled. More importantly, the controls over the interface can be abstracted from the actual hardware implementation. Virtualization technology should also enable a gradual evolution to this model from what (hopefully) will one day be called "legacy SMM mode":-)

4.11 Non-Restartable Instructions

This is basically "just" a cleanup of existing microcode, and very careful placement of the SVM intercepts. Refer back to Section 2.1.6 for a description of the problem. The intent is that all guest instructions (except possibly task switches) should be restartable after any intercept.

4.12 External Access Protection

By securing the virtual address translation mechanism, trusted memory is protected from direct access by untrusted code running on the processor. But to fully protect trusted memory it is also necessary to control external device accesses to memory. Peripheral devices access physical memory directly, bypassing the virtual memory mechanisms, hence a separate protection mechanism is required.

This mechanism prevents untrusted code from attacking trusted memory indirectly through something like a simple plug-in device designed to circumvent memory protection, or by subverting an existing peripheral device, as may be possible via writable firmware. Protection against unauthorized device access is accomplished via control logic added to the system memory controller which governs any external access port (e.g. PCI or HyperTransport interfaces).

IMPORTANT: for proper security, the platform needs to allow the VMM to prevent peer-to-peer device transactions between devices in different protection domains. VMM software needs to be aware of those platform mechanisms, and any constraints they may impose on the granularity at which device ownership can be granted to different protection domains.

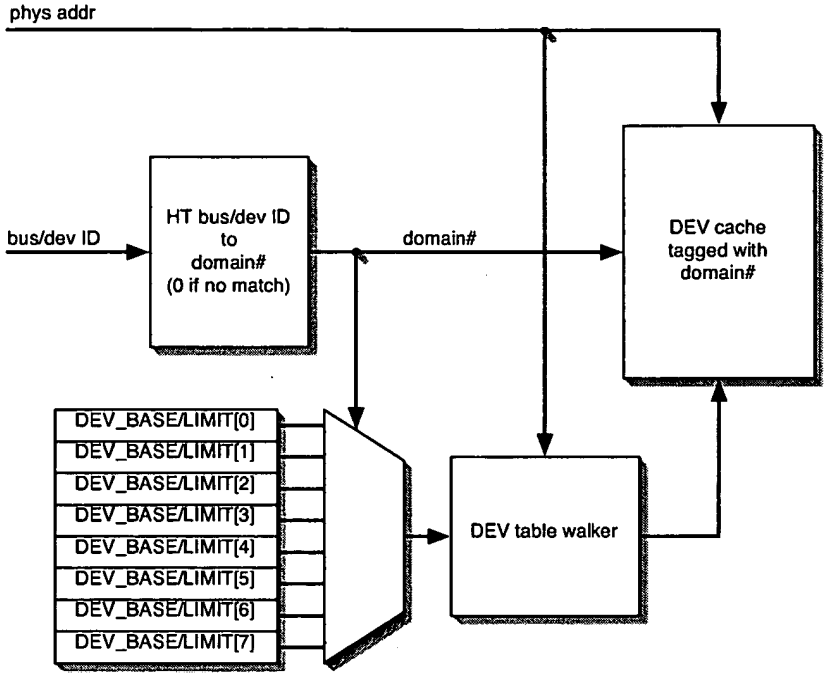
4.12.1 Device IDs and Protection Domains

With VM support, the memory controller provides a small number (8 to 16) of *protection domains*; each protection domain has associated with it a *Device Exclusion Vector* (DEV). For this implementation, devices are identified by an HT bus/device ID, and the memory controller contains an associated lookup table of fixed size (TBD) that maps device IDs to a protection domain; the protection domain number in turn selects the DEV that specified the device's access rights to physical memory.

NOTE: in the future, we will likely support a mechanism that maps PciExpress deviceIDs to protection domains; it is all but certain that a slightly different structure will be required than for HT bus/dev IDs. As a result, the code that sets up the deviceID-to-domain maps will be implementation dependent.

4.12.2 Device Exclusion Vectors

Figure 4. Memory Controller DMA Checking



A DEV is a contiguous array of bits; each entry in the DEV corresponds to one page in physical memory. The DEV is referenced by access control logic in the system memory controller for any reference on an external port (i.e. any port other than the processor port). If a bit in the DEV is set to a 1, the memory controller shall inhibit all external bus-master or DMA device cycles targeting the associated physical page. Read accesses by devices to DEV protected addresses shall return all 1's and write access shall terminate without affecting the contents of memory. Any page that is beyond the programmed range of the DEV is freely accessible by external devices.

The DEVs are set up and maintained by the VMM, and is referenced by the memory controller logic in the course of enforcing device access protections. Reads of a DEV by memory controller hardware are done using a UC memory type, hence the DEV should be mapped as UC memory for updating by software. (If the performance of code manipulating the DEV is a concern, it may be mapped as WB memory as long as any modified blocks are flushed from the caches via the CLFLUSH instruction before any device accesses to the updated entries would occur.) The physical address of the base of a DEV must be 4KB-aligned, and is stored in one of the DEVBASE registers, which are in the DEVCTL PCI Configuration Space function block in the memory controller (see XXX for details). 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.

When a read or write request is received on an external (non-CPU) memory port, the memory controller must map the device ID to a protection domain number, which in turn selects the DEV defining the access permissions for the device. The memory controller then checks the memory address against the DEV contents by indexing into the DEV with the PFN portion of the address (bits 31:12). (Note that the PFN is used as a bit index within the DEV.) If the bit read from the DEV is set to one, the memory controller must inhibit the access by returning all ones for the data for a read request, or suppressing the store operation on a write request. An error response, such as Target Abort, may be returned to the requesting device. The DEV must be located in an uncacheable (with respect to processor caches) or write-through memory region in order for DEV updates to be visible to device access protection logic. This may be configured either via an MTRR or via the page cacheability attributes (PCD, PWT) of the page table entries that map the DEV.

IMPORTANT: the DEV table is not consulted by the CPU, even in real mode. This is different from what was done in SEM. The VMM is responsible for never letting the guest enter (physical) real mode; paged real mode must be used instead.

4.12.3 DEV Control Registers

Control registers for the DEV protection mechanism are contained in a new function block in the memory controller's PCI Configuration space. This function block defines the following registers:

Table 9: DEV Control Registers

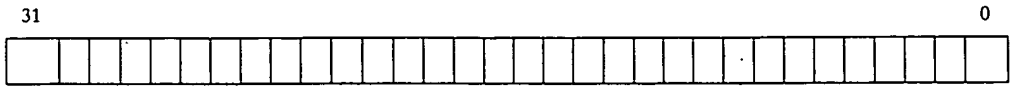
DEV_VER	Version / Implementation Information
DEVCR	DEV Control Register
DEVLOG	DEV Access Logging Register
DEVBASE[0..n]	DEV Base Registers; the number of registers is implementation dependent.

As these registers are accessed via PCI config space (ports 0CF8-0CFF), they may be secured from unauthorized access by software executing on the processor by appropriate settings in the SVM I/O Protection Bitmap. They are also protected by the memory controller from external access via peer-to-peer device transactions, as described in Section 4.12.6, Protection of I/O Space from External Access. The DEV protection hardware is enabled by setting the DEVCR DEVE (bit 0).

4.12.3.1 DEV_VER Register

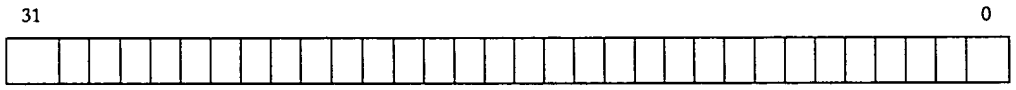
Read-only register; holds implementation specific information such as the number of protection domains supported.

Table 10: Format of DEV_VER Register (in PCI Config Space)



4.12.3.2 DEVCR Register

Table 11: Format of DEVCR Register (in PCI Config Space)



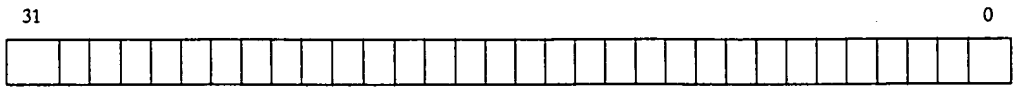
4.12.3.3 DEVLOG Register

Table 12: Format of DEVLOG Register (in PCI Config Space)

4.12.3.4 DEVBASE Address/Limit Registers

The DEV Base Address registers point to the physical address of a DEV table; the register format is as follows: *TODO*: split into BaseHi and BaseLo? Where to place the limit?

Table 13: Format of DEV Base Registers (in PCI Config Space)



4.12.4 Memory Controller DEV Caching

To avoid incurring the overhead of referencing the actual DEV in DRAM for each device access to memory, some amount of DEV permission information should be cached in high-speed logic in the

memory controller for quick reference. This may be as simple as recording the address of just the most recent permitted access in anticipation of subsequent accesses to the same page (such as for a DMA block transfer), or as complex as caching various portions of the DEV contents, tagged with the protection domain (essentially, a "protection lookaside buffer").

The means by which such caching is done is implementation-dependent, but in all cases, hardware must invalidate any such cached information whenever software sets 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 memory controller will automatically clear this flag when the flush operation is complete. After setting this flag, software should monitor it until it has cleared in order to synchronize DEV updates with subsequent activity.

4.12.5 Unauthorized Access Logging

Any attempted access by devices to DEV-protected memory shall be logged by the memory controller in the Device Log (DEVLOG) status register for possible inspection by the SK. The information logged shall consist of:

- the target address
- a Valid flag to indicate that such an access attempt has been logged, and
- an Overflow flag which indicates that subsequent access attempts have occurred which have not been logged.

An access may be logged only when the Valid flag is clear, and the act of logging an access shall set the Valid flag. DEV-inhibited accesses that occur when the Valid flag is set will cause the Overflow Flag to be set. The SK may periodically read the DEVLOG register to check for unauthorized access attempts. The SK will typically clear the Valid bit by writing zero to it after reading the register, to re-enable logging.

4.12.6 Protection of I/O Space from External Access

Major aspects of memory controller functionality are configured via control registers that are accessed via PCI configuration space. Since this is potentially accessible via device peer-to-peer transfers, the memory controller must block access to this space from anything other than the CPU. Accesses from the CPU are controlled by the SVM I/O Protection Bitmap. Details of this peer-to-peer protection mechanism are beyond the scope of this document, but are provided in the SEM Platform Specification.

4.12.7 Secure Initialization Support

The memory controller contains additional logic that operates in conjunction with the SKINIT instruction to support the secure and verifiable startup of a Security Kernel loader (SL). (See

Section 4.13.5 for detailed operation of SKINIT.) This memory controller logic includes the following:

- SL_DEV_BASE address, and associated SL_DEV enable bit (SL_DEV_EN) in DEVCR. SL_DEV_BASE points to a naturally-aligned 64KB region of physical memory. This register is not directly accessible by software – it is loaded only by the SKINIT instruction through a special protocol, and simultaneously enabled via SL_DEV_EN, which is automatically set. It is not readable. When this is active, the 64KB region defined by SL_DEV_BASE is protected from external access as if it were protected by the DEV. This mechanism also protects this region against access via GART-translated addresses, ensures that CPU accesses to memory are not remapped by the GART, and also blocks device accesses to PCI Configuration space.

  The DEV logic is typically not operational when SL_DEV_BASE is enabled, so this provides protection for a Secure Loader image in memory, allowing it to, among other things, set up full DEV protection. SL_DEV_EN may be cleared by the SL simultaneously with enabling the DEV logic.
- Support for secure transfer of the SL image from the processor to an external device (SSP), with a special protocol (unique control, address and/or commands) that is used only by the SKINIT instruction. This includes explicit start-of-transfer and end-of- transfer indications which bracket a variable number of data transfer cycles (anywhere from none to 64KB worth). The protocol does not allow any other device to generate these start, end, and data transfer indications.
- Support for multiprocessor synchronization. The start-of-transfer command must return an indication to the processor of whether or not all other processors are in APIC INIT state (i.e. waiting for an APIC Startup IPI).
- SL_DEV support on multiple-memory-controller systems. The SL_DEV_BASE address is distributed to and enabled on all memory controllers simultaneously, coincident with or prior to the setting of the start-of-transfer status indication.

4.13 Secure SK Startup with SKINIT

One of the main features of SVM is that SVM protections may be reliably enabled after the system is up and running – there is no requirement to change the typical x86 platform boot process. SVM protections will typically be turned on by a system software component that, for the purposes of this document, is referred to as the SVM driver. The SVM driver will typically be independent from the OS kernel, but run in kernel mode, much like any device driver.

The key to reliably establishing SVM protections from within the unprotected system environment of the typical x86 platform is the SKINIT instruction, which is intended for use by the SVM driver. 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. It also copies the Secure Loader executable image to an external device, such as a Secure Service Processor (SSP) for verification using unique bus transactions that preclude SKINIT operation from being emulated by software in a way that the SSP could not readily detect. (Detailed operation is described in Section 4.13.5.)

The Secure Loader will typically initialize SVM hardware mechanisms and related data structures, validate the SK executable image, and start execution of the SK. The SK in turn takes control of the address translation tables, control registers and the SVM hardware mechanisms, and then returns to the normal OS which resumes operation where it left off, but with full SVM protections in place.

Exact details of SVM driver and SL operation are for the most part dependent on various OS, SL and SK characteristics, and are mostly up to software. There are specific hardware requirements for the SL image, however, as described in Section 4.13.1.

[NOTE: change from SEM]
The SKINIT instruction is privileged and can only be executed at CPL0. It may be executed only at CPL 0, and the VMM will typically intercept the SKINIT instruction in guests. SKINIT is intended to be used primarily in normal mode prior to SVM being enabled, although it may be used by the SK to re-initialize the system if desired. Operation is the same in either mode.

4.13.1 Secure Loader Image

The Secure Loader image contains the code and initialized data sections of a complete implementation of a Secure Loader. It must contain all the code and initial data needed to initialize and start a Security Kernel in a completely safe manner. This includes setting up DEV protection for memory allocated for SL and SK use. It is loaded by the SVM driver into a region of memory called the Secure Loader Block (SLB, described below) and can be no larger than 64KB in size. (The 64 KB limit is enforced by hardware; in practice, for reasons described below, the maximum size may be somewhat less.) The SL image is defined to start at byte offset 0 in the SLB.

The first word (16 bits) of the SL image must contain an unsigned offset into the SL image of the SL entrypoint; the second word must contain the length of the image, in bytes, with the maximum length being 65535. These two values are used by the SKINIT instruction. The layout of the rest of the image is determined by software conventions. The image will typically include a digital signature for validation purposes; the hash for this must include the entrypoint and length fields. SKINIT transfers the SL image to the SSP for validation prior to starting SL execution; see Section 4.13.5 for further details of this transfer. The SL image for which the hash is computed must be ready to execute with no fixups necessary.

The SL must be written to execute, at least initially, in flat 32-bit protected mode with paging disabled. Note that 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.

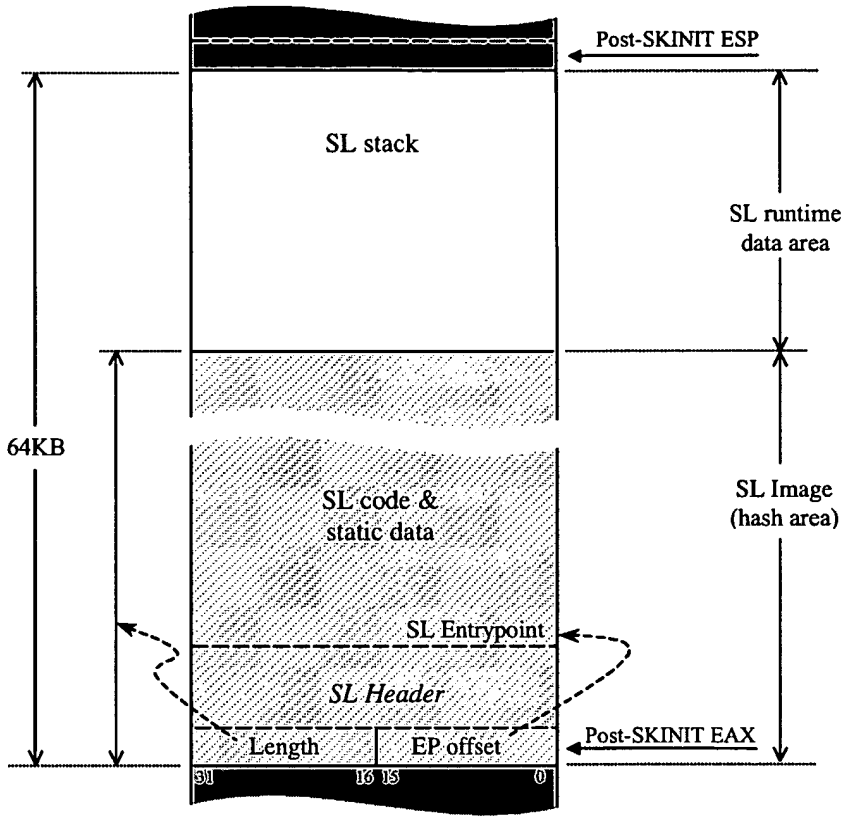
4.13.2 Secure Loader Block

The Secure Loader Block is a 64KB range of physical memory which may be located at any 64KB-aligned address below 4GB. This memory may be allocated by the SVM driver or set aside anytime beforehand. The SVM driver must load the SL image into the SLB starting at offset 0 anytime 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.

Memory between the end of the SL image and the end of the SLB may be used immediately upon entry by the Secure Loader 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 for this (EAX + 10000h).

Figure 5 illustrates the layout of the SLB. This shows where EAX and ESP point after SKINIT execution. Labels in italics indicate suggested uses; other labels reflect required items.

Figure 5. SLB Example Layout



4.13.3 Secure Service Processor

The Secure Service Processor, or SSP, is an essential part of full SVM system initialization. This device is attached to an LPC link off the system's I/O Hub. It recognizes special SKINIT transactions, receives the SL image sent by SKINIT and verifies the signature. Based on the outcome, it decides whether or not to cooperate with the SL or subsequent SK. The SSP typically contains sealed storage containing cryptographic keys and other high-security information that may be specific to the platform. SSP hardware and software interface requirements may be found in the SEM Platform Architecture and SSP API specifications.

4.13.4 System Interface, Memory Controller and I/O Hub Logic

SKINIT requires special support logic in the processor's system interface unit, the internal or external memory controller(s) and the I/O Hub to which the SSP is attached. SKINIT uses three special transactions that are unique to SKINIT, along with this support logic, for securely transmitting the

SL Image to the SSP for validation. These transactions are called HASH_START, HASH_DATA, and HASH_END.

The use of this special protocol allows the SSP 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.

The required support logic consists of the following functions:

System interface unit:

- A one-bit HASH_START_STATUS flag. This indicates success or failure of the HASH_START transaction on an MP system. This flag is not directly visible to software.
- A one-bit HASH_START_SEEN flag. This indicates that a HASH_START transaction was issued by another processor on the system. This flag is not directly visible to software.
- Logic for broadcasting the HASH_START transaction to all other memory controllers in the system.
- A means for checking the APIC state of all other processors in the system.

Memory controller:

- An SL_DEV_BASE address register and SL_DEV enable bit which, when active, protects the 64KB SLB region from external access as though it were protected by the DEV (and including protection against GART-translated accesses by external devices). It also prevents CPU accesses to memory from being remapped by the GART.
- Logic to load the SL_DEV_BASE and enable SL_DEV protection with an SL Image base address supplied by the HASH_START transaction.
- Logic to route the HASH_START, HASH_DATA and HASH_END transactions to the I/O Hub.

I/O Hub:

- Logic to route the three SKINIT transaction types over the LPC bus to an SSP.

4.13.5 SKINIT Operation

SKINIT takes as its only input operand, in EAX, the physical base address of the SLB, and performs the following steps:

1. [NOTE: change from SEM] Generate a #GP exception if the CPL is not 0, check for SVM intercept.
2. Clear any installed microcode patches. (Up to and including this point, the SKINIT microcode is not patchable. This effectively makes the rest of the SKINIT instruction unpatchable as well.)

3. Reinitialize processor state in the same manner as for the INIT signal, then put the processor into flat 32-bit protected mode (with paging off) rather than leaving it in real mode. CR0.PE is set to 1; CS and SS selectors are set to 0008h and 0010h respectively, and CS and SS base, limit and attribute registers are set to the same values as for SMCALL (*TODO*). DS, ES, FS and GS are left as 16-bit real mode segments, and the SL must reload these with protected mode selectors with 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 registers are cleared except for EAX, which retains its value, 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 operand and control registers. Most MSRs also retain their values, except those which could compromise SVM protections; the EFER MSR however is cleared. The VM_CR.DPD flag is unconditionally set, disabling certain hardware debug features.
4. Form the SLB base address by clearing bits 15:0 of EAX (EAX is updated).
5. Send the HASH_START transaction with the SLB base address to the memory controller(s), which uses this to initialize and enable the SL_DEV protection mechanism. The local memory controller also passes this on to the I/O Hub, which in turn passes it on to the SSP. The SSP then knows to expect an SL image transfer for validation. (Note that the SLB base address isn't necessarily passed on, just the command.)
6. Wait for a response from the system interface indicating completion of the HASH_START transaction, and then check the HASH_START_STATUS flag.
7. If HASH_START_STATUS indicates success, the SL image is read from memory and transmitted to the memory controller using a series of HASH_DATA transactions to a fixed device address. The memory controller passes these on to the I/O Hub, which routes them to the SSP. The size of these transfers (i.e. byte, dword, qword) is implementation-dependent, and the last transfer may include some number of bytes beyond the end of the SL image if the image length is not a multiple of the transfer size. The SSP must use the length field in bytes 2 and 3 of the image to determine the exact length for the hash function.
   If HASH_START_STATUS indicates failure, this data transfer is skipped. (This situation only applies to multiprocessor operation, which is described below.)
8. Send a HASH_END transaction, which is routed to the SSP. This signals the SSP to complete the hash and verify the signature. If no data was transmitted in step 7 due to the HASH_START_STATUS flag indicating failure, the SSP will conclude that no valid SL was started.
9. Clear the Global Interrupt Flag. This disables all interrupts, including NMI, SMI and INIT, ensuring that the subsequent code can execute atomically. [NOTE: **change from SEM**] The SL and/or VMM are responsible for ensuring that stgi and clgi instructions are intercepted, i.e., not available to guests. If the processor enters the shutdown state (due to a triple fault for instance) while GIF is clear, it can only be restarted via RESET#.
10. [NOTE: **change from SEM**] Mask A20M# via the `vm_cr.dis_a20m` bit.
11. Update ESP register to point to the first byte beyond the end of the SLB (SLB base + 65536), such that the first item pushed onto the stack by the SL will be at the top of the SLB.
12. Add the unsigned 16-bit entrypoint offset value from the SLB to the SLB base address to form the SL entrypoint address, and jump to it.

Note that the validation of the SL image by the SSP is a one-way transaction as far as SKINIT is concerned. It does not depend on any response from the SSP after transferring the SL image before jumping to the SL entrypoint, and initiates execution of the Secure Loader unconditionally. Because of the processor initialization it performs, 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: In order to make special hardware debug features available such as HDT, a debug version of the SL should clear the VM_CR DPD flag immediately upon entry via WRMSR.

I/O and PCI Configuration Space access: I/O SMI traps, which may occur on selected I/O port or PCI Configuration Space locations per the I/O Trapping facility in the processor (typically for the purpose of emulating certain PCI Configuration space registers) are suppressed when GIF is clear. This ensures that the Secure Loader can access actual hardware registers without intervention by untrusted SMM-based code.

4.13.6 SL Abort

If the SL determines that it cannot properly initialize a valid SK, it must cause GIF to be set in order to re-enable normal hardware interrupt and debug trap functionality. In addition, certain debug features are disabled by SKINIT when it sets VM_CR.DPD, and these should be re-enabled by clearing DPD via a normal MSR write.

4.13.7 Secure Multiprocessor Initialization

The SVM initialization process for a multiprocessor system relies on system interface logic that is activated by the SKINIT instruction, in conjunction with existing APIC MP support features that must be used by the SEM driver and the Secure Loader or SK.

4.13.7.1 Hardware support for MP initialization

The following standard APIC features are used for SVM MP initialization (with minor modifications as noted):

- 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, which 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 SVM initialization, as described below.

The hardware specific to SKINIT consists of (in each processor):

- Logic to indicate to the BSP, in response to a HASH_START command, whether or not the processor is in APIC INIT state.
- A HASH_START_SEEN flag that indicates the receipt of a HASH_START command sent out from the BSP when SKINIT executes.

A Startup IPI normally causes an AP to start execution at a location provided along with the IPI. To support MP SVM startup, each AP will additionally clear its GIF and set its local VM_CR.DPD flag if, and only if, on receipt of a Startup IPI its local HASH_START_SEEN flag is asserted. (GIF would have been initialized to 1 by a preceding INIT IPI, or an INIT or RESET# assertion.) The HASH_START_SEEN flag is also cleared. All other aspects of Startup IPI behavior remain unchanged.

4.13.7.2 Hardware operation and Software requirements for SVM MP initialization

The SVM driver 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 an OS kernel function, or via SVM driver threads running on the other processors). Once it has confirmed that all APs are idle, the driver must issue an Init IPI to all APs, then wait for its local APIC Busy indication to clear. This places the APs into a halted state which is responsive only to a subsequent Startup IPI (although they will still respond to snoops for cache coherency). The driver may execute SKINIT anytime 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 HASH_START command.

4.13.7.2.1 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 to the APs which is specified by the software that issues the IPI. This vector provides the upper 8 bits of a physical page address, therefore the startup code must reside in the lower 1MB of physical memory, with the entrypoint at offset 0 on that particular page. The SL (or SK) may need to temporarily move a portion of memory in order to load the AP startup code at an appropriate location, and restore it before resuming normal system operation.

In response to the Startup IPI, the APs start executing at the specified location in 16-bit real mode. This AP startup code must initialize the VM-related MSRs on each processor to appropriate values (as determined by the SL or SK). [NOTE: change from SEM] It must also set GIF and re-enable

interrupts, and restore the pre-SKINIT system context (as directed by the SL or SK executing on the BSP), before control of the processor is handed back to the OS.

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. (SKINIT would execute, but would likely abort the SL Image transfer since the BSP would not be in APIC Init state, and confuse or reset SSP.)

4.13.7.2.2 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 anytime 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.

4.13.7.2.3 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 OS:

- [NOTE: **change from SEM**] The BSP and all APs that responded to the Startup IPI must restore the GIF on each processor for normal interrupt operation. They must also clear VM_CR.DPD, via WRMSR, to restore hardware debug functionality disabled by the Startup IPI.
- 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.
- After all AP-specific clean-up has completed, any memory contents in the lower 1MB that were set aside by the SL to make room for AP Startup IPI code must be restored.

Any secure context created by the SL which should not be exposed to untrusted code or device activity should be cleaned up as appropriate before these steps are taken.

4.14 Functional Changes via Microcode Patch or JTAG

Functionality of the processor (and any logic integrated with the processor on the same die, such as a memory controller) may be subject to modification through microcode patching or JTAG scan functions. Processors that support SVM must not allow software-based attacks that would compromise SVM protections using these mechanisms, and as much as possible must limit hardware-based attacks.

4.14.1 Microcode patch mechanism

The microcode patch mechanism is controlled by MSRs and hence may be fully controlled by the VMM, once the VMM is running. However, additional precautions are needed to prevent SVM functionality from being compromised before the VMM gains control. These are:

1. Microcode patches are automatically removed by the SKINIT instruction when it executes, as described in Section 4.13.5, SKINIT Operation.
2. It is not possible to patch the SKINIT instruction itself to skip over or disable the microcode patch removal step, hence it is not possible to modify any aspect of SKINIT operation via microcode patches.
3. It is not possible to patch microcode in a way that would allow the special SL image bus transactions to be generated without executing the SKINIT instruction. If any microcode patches are required for system operation under SVM, they must be validated and loaded by the Secure Loader as part of the secure SVM initialization procedure. Once SVM is set up, the patching mechanism must be made accessible only by the VMM via appropriate settings in the MSR protection bitmap.

4.14.2 JTAG operation

The processor's JTAG port supports manufacturing test functions via control logic scan, hardware debug functions via the Hardware Debug Tool (HDT), and manufacturing-time product configuration via programmable fuses.

Scan: Scan functionality cannot be programmatically disabled. However, the possible use of scan capabilities for circumventing SVM protections greatly exceeds the cost-of-attack model that SVM is designed to cover.

Product configuration fuses: these are not permanently reprogrammable after manufacturing, however there is a temporary fuse override capability that may be used in systems for certain test purposes. This override capability is disabled by the VMCR.DPD bit whenever this bit is set to 1.

HDT functionality: HDT operation may be completely disabled by setting VMCR.DPD.

4.15 New MSRs

Many of these MSRs are only there to make otherwise hidden VM-related bits visible to s/w (debug support request from processor groups). Implementation concerns may require that bits be assigned to different MSRs (or replicated in multiple MSRs.)

- INTCTL (r/o) — various interrupt control bits, virtual and physical
  - phys_if:1 — the bit that masks INTR interrupts. Written to the same value as eflags.if, whenever eflags.if is changed while the INTR intercept is zero.
  - GIF:1 — global interrupt flags. Written by stgi and clgi instructions. Resets to "1".
  - v_irq:1 — "virtual interrupt pending" bit. Written by SVM, always cleared on vm-exit. resets to "0".
  - v_intr_prio:8 — priority of a pending virtual interrupt. Written by SVM.
  - v_intr_vector:8 — vector for pending virtual interrupt. Written by SVM.
  - v_tpr:8 — virtual TPR; written by SVM; accessed by references to CR8 as long as the INTR intercept is "1".
- VM_INTERCEPT0, VM_INTERCEPT1 (r/o): bitvectors indicating what operations to intercept; written by SVM, always cleared on vm-exit. Reset to "0".
- VM_CR (r/w) — SEM_CR carried forward.
  - svm_ena:1 — enable bit for the new instructions which otherwise cause #UD exceptions.
  - rve:1 — root vector check enable. Same as in SEM.
  - dpd:1 — debug disable. Same as in SEM.
  - r_init:1 — when in host mode, INIT gets turned into an #SX interrupt if this bit is set. Same functionality for intercepting INIT as in SEM. Resets to "0".
  - dis_a20m — if set, disables A20 masking.
- SMM_CTL (w/o) — software control over SMM signals.
  - smm_enter:1 — write this bit to "1" to enter SMM (map SMRAM areas, block NMI and various other interrupts, ...)
  - smm_ack:1 — write this bit to "1" to externally signal that the processor is entering SMM
  - smm_exit:1 — write this bit to "1" to exit SMM (unmap SMRAM areas, reenableNMI and various other interrupts, ...)
  - smm_nak:1 — write this bit to "1" to externally signal that the processor is no longer in SMM.
- VM_MSRPM_BASE (r/o) — same function as in SEM, but loaded by SVM (hence a r/o MSR). Note: the format of the table has changed wrt. SEM, so we can separately intercept MSR reads and MSR writes.
- VM_IOPM_BASE (r/o) — same function as in SEM, but loaded by SVM (hence a r/o MSR).
- IOIO_RIP, IOIO_TYPE, IOIO_RCX, IOIO_RSI, IOIO_RDI, IOIO_SMI_ON_IO (r/o) — all the information that IN/OUT instructions save (for SMM I/O instruction restart). These will be useful not only for simulating SMM, but also will speed up general IOIO intercept handling.

- VM_VMCB_PA (r/o) — SVM saves the physical address of the VMCB here; on vm-exit, the guest state is written back to this address.
- VM_HSAVE_PA (r/w) — the physical address of a block of memory where SVM saves host state, and where vm-exit reloads host state from. The VMM software is expected to set up this register before issuing the first SVM instruction.
- VM_RUNLIMIT (r/o)
- VM_TSC_OFFSET (r/o)
- VM_CURR_ASID (r/o)
- IGNNE (r/w) — 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).

Table 14 below summarizes the new MSRs.

Table 14: Secure-VM New MSRs.

Name	Address *(tentative)*	Access	Description
INTCTL	C001_0114	r/o	Status of virtual and physical interrupt controls.
IOIO_RCX	C001_0115	r/o	Information saved by IN/OUT instructions for SMM I/O restart; made explicitly visible to the VMM via these MSRs.
IOIO_RDI	C001_0116	r/o	
IOIO_RIP	C001_0117	r/o	
IOIO_RSI	C001_0118	r/o	
IOIO_SMI_ON_IO	C001_0119	r/o	
IOIO_TYPE	C001_011A	r/o	
SMM_CTL	C001_011B	w/o	Explicit control over SMM state and signals.
VM_CR	C001_011C	r/w	SVM enable, security-related mask bits.
VM_CURR_ASID	C001_011D	r/o	Current ASID.
VM_HSAVE_PA	C001_011E	r/w	Physical address of host state-save area.
VM_IOPM_BASE	C001_011F	r/o	Physical address of current IOPM bitmap.
VM_MSRPM_BASE	C001_0120	r/o	Physical address of current MSRPM bitmap.
VM_VMCB_PA	C001_0121	r/o	Physical address of last VMCB loaded by SVM
IGNNE		r/w	Set the processor-internal IGNNE state.

4.16 New Instructions

Table 15, below, lists the instructions added for Secure VM. NOTE that some of the opcodes are still tentative. If `vm_cr.en_svm` is zero, all of these instructions cause #UD exceptions.

Table 15: Secure-VM New Instructions

Instruction	Encoding	Operation	Notes
`vmmcall`	`0F 01 F9`	if not CPL-0: #GP if intercepted: vm-exit else no-op	
`SVM [eax]`	`0F 01 FA`	if not CPL-0: #GP if intercepted: vm-exit else world-switch to guest (see Section 4.1)	`eax` holds physical address of VMCB; no #PF can occur.
`stgi`	`0F 01 FB`	if not CPL-0: #GP if intercepted: vm-exit else set Global Interrupt Flag	
`clgi`	`0F 01 FC`	if not CPL-0: #GP if intercepted: vm-exit else clear Global Interrupt Flag	
`skinit [eax]`	`0F 01 FE`	if not CPL-0: #GP if intercepted: vm-exit else secure init (see Section 4.6)	
`vmload [eax]`	`0F 01 FD` (tentative)	if not CPL-0: #GP if intercepted: vm-exit else load guest state from VMCB	`eax` holds physical address of VMCB; no #PF can occur.
`vmsave [eax]`	`0F 01 FF` (tentative)	if not CPL-0: #GP if intercepted: vm-exit else save guest state to VMCB	`eax` holds physical address of VMCB; no #PF can occur.

5. How to Virtualize...

In this section, we describe in detail how VMM software might use the proposed h/w to virtualize various aspects and modes of an x86 system.

5.1 Intercepts

With the proposed h/w support, VMMs need *not* use the common technique of ring compression; instead, guest code will execute at its intended CPL. Interception of guest operations is performed by setting appropriate bits in the intercept field of the VMCB.

5.2 World Switch via SVM

The VMM will use the SVM instruction to switch key host/guest state atomically; additional state save and restore is done by the VMM in software, and as necessary (e.g., dependent on the intercept code seen after a vm-exit).

5.2.1 VMM Main Loop

The VMM main loop will likely look somewhat like this:

```
    // assume eax points at VMCB for current guest
slow_resume:
    gif := 0 // turn of interrupts since we won't be interrupt-safe for a while
    save any host state that VMM s/w cannot just reconstruct
    VMLOAD [eax] // load guest hidden state
    in s/w, load guest state not loaded by either VMLOAD or SVM
quick_resume:
    // SVM will atomically set gif:=1, and save/restore minimal state
    SVM [eax]
    // after exiting SVM, GIF will once again be zero
    switch (vmcb->exitcode)
       case simple_intercept:
          restore minimal host state
          handle intercept
          restore minimal guest state
          goto quick_resume;
    case complex_intercept: // including pending physical interrupts, or new guest
          VMSAVE [eax] // save all guest hidden state
          save remaining guest state in software
          load up all relevant host state (no need to handle hidden state here)
          gif := 1 // now safe for interrupts again
          handle intercept (or take physical interrupt here)
          goto slow_resume;
    }
```

State switched by the VMM in software will include the following

- all GPRs (except for `eax`, which is switched by SVM)
- all FPRs (but see note in Section 5.2.2)
- `cr2` — no need to save this in h/w; the VMM need only save the guest's value if either it expects to receive a #PF itself, or if it decides to switch to another guest.
- any additional control registers / MSRs that the VMM is not virtualizing entirely in software.

The details would of course depend heavily on the architecture of the specific VMM; the key is that we provide the h/w support to (a) perform a minimal but still "useful" world switch in SVM, and (b) access any hidden guest state that can reasonably need to be switched.

5.2.2 Lazy FP State Save

Since FP state is large, the VMM may want to save and restore FP state lazily (*across* guests, just as an O/S might do *within* a guest). To do so, the VMM needs to intercept all FP use while running the guest. When the intercept triggers, the VMM can save the FP state, load the guest FP state, and resume the guest, now without the intercept.

The same effect can be achieved by virtualizing `cr0.ts` in the guest, but then the `clts` instruction needs to be intercepted, which would probably make lazy FP save/restore *within* the guest slower than eager save/restore.

5.2.3 FERR and IGNNE

Existing K8 processors already include a mechanism to *emulate* the IGNNE signal internally; using this functionality as a base, the VMM can emulate the legacy FP-exception handling as follows:

- enable IGNNE emulation via the HW_CR MSR. This means the external IGNNE pin is ignored by the processor, and the processor maintains its own internal copy of the signal by monitoring OUT instructions to port 0xF0.
- mask FP exceptions in the APIC or PIC, so that the FERR pin cannot cause any physical interrupts to the processor.
- enable the `ferr_freeze` intercept; when this intercept triggers, the VMM can inject a virtual interrupt into the guest.
- the guest can be given direct access the internal IGNNE state (via port 0xF0); when switching to a different guest, the VMM can save/restore the IGNNE signal for the outgoing and incoming guests via the IGNNE MSR. [Note: this can be done without the IGNNE MSR, but then accesses to port 0xF0 would have to be trapped to the VMM, and restoring the IGNNE state for an incoming guest gets more complicated.]

5.3 Memory and Paging

Generally speaking, the VMM builds a shadow page table (eagerly, or on demand). We first describe the general support for maintaining shadow page tables. We then discuss related topics, such as how to virtualize the guest PAT, MTRRs, IORRs, the legacy region and SMM memory remapping. Special treatment for x86 operating modes is described in Section 5.4.

To maintain shadow page tables, the VMM needs to intercept the following operations:

- `cr0` writes — the WP, NW, CD and PG bits can change virtual address translation, hence writing them requires updating the shadow page tables.
- `cr3` reads and writes — clearly, on a cr3 write, the shadow page table needs to be updated. On a cr3 write, the VMM must deliver the value last "written" by the guest, as opposed to the actual value loaded into cr3 (which would be the address of the shadow page table).
- page faults — guest page faults may occur because of a "real" page fault in the guest's page table, because the shadow page table has not been fully filled in yet, or because the host has unmapped a guest physical page. Only in the former case does the exception get reflected to the guest; in the other cases, the shadow page table must be updated (and/or the host must remap a guest physical page).
- `cr2` writes — as the previous item shows, not all of the page faults incurred by the guest are meant to be visible to the guest. However, all of them write `cr2`. Therefore, the VMM must intercept guest's (explicit) writes to `cr2` so that it always knows the expected value. On a page fault that is *not* passed to the guest, the VMM must restore the guest's view of `cr2` to that value.
- `cr4` writes — the PGE, PAE, and PSE bits affect paging, and require the shadow page table to be updated. Assuming that this register is written infrequently, a selective intercept (triggering only when the bits listed here change) may not be necessary.
- `efer` writes — same reasoning as cr4.
- `pat` writes and possibly reads — the host may wish to maintain its own copy of pat.
- `invalpg` — the shadow page table needs to be updated, and guest TLB entries need to be invalidated.

Note that when the *host* changes its address map (which is, after all, factored into the shadow page table), potentially all guest shadow page table entries (and TLB entries) need to be invalidated. SVM provides the necessary mechanisms; the details of how VMM s/w tracks such changes (as well as policy issues regarding more aggressive management of shadow page tables) are beyond the scope of this document.

5.3.1 Legacy Region

Virtualizing the legacy region can be done with help from the s/w that manages shadow page tables. To do so, writes to the MTRRs that map the legacy region need to be intercepted (MSR intercept), and the shadow pages need to be updated in accordance with the settings for the region. For example

- read from ROM, write to ROM or MMIO: map the guest access to a host page containing the guest's (virtual) ROM. Unless this is a real (physical) ROM, make the page read-only. Page faults due to writes need to either be ignored (write to ROM), or dispatch to the I/O device emulation code (write to MMIO).
- read from RAM, write to ROM or MMIO: similar to the above.
- read from ROM, write to RAM: ditto.
- read from RAM, write to RAM: just map a plain read-write host page.

5.3.2 IORRs, MTRRs, (Virtualized) SMM remapping

These can all be handled in a manner similar to the legacy region. As the shadow page table is filled in, check whether the guest physical address needs special handling, and adjust the permissions, the host physical address, or memory type in the shadow page table accordingly.

For example, when not in SMM, some SMM RAM regions would be marked as not-present in the shadow page table, and only revealed to the guest while in (virtual) SMM. This of course only applies to virtualized SMM inside a guest; for running physical SMM inside a container, see XXX.

Note that all changes to IORRs and MTRRs, as well as entering and exiting SMM must be intercepted, so the VMM can adjust the shadow page tables accordingly.

5.3.3 Memory Types

Memory types in the guest, as defined by mtrrs, page table / cr3 bits and the guest `pat`, can be emulated by constructing the shadow page table appropriately. Specifically, start with the GV address, and walk the guest page table to determine the GP address. Determine the corresponding (guest) memory type T_g(from all various guest sources). Set the memory type bits in the shadow page table entry such that the resulting pat index maps to a host memory type $T_h == T_g$.

Note: there is a potential problem if the host's own page tables access the corresponding memory with a different memory type than the guest (via the shadow page tables) — some x86 processors are not well-behaved under those circumstances, and may deliver unpredictable results. In such cases, the VMM may want to adjust the memory types in either or both host and shadow page tables so they are in agreement.

5.4 Major X86 Operating Modes

In this section we describe additional work that the VMM must do in order to virtualize the major operating modes of the x86: protected mode (with paging enabled or disabled), virtual-x86 mode, real mode, and SMM.

5.4.1 Protected Mode, Paging Enabled

This is the mode we expect to spend most time in, and the h/w support is optimized for it. In the discussions that follow, protected mode with paging enabled serves as the baseline.

5.4.2 Protected Mode, 32-bit PAE Mode

Kevin expressed some concern about virtualizing legacy (32-bit) PAE paging mode — in the current implementation, the four top-level PDPEs are cached in the CPU every time cr3 (and/or cr4?) is written. Table walks never read the PDPEs in memory and instead use the cached values, this allows the in-memory value to become inconsistent with the hidden on-chip state, and guest s/w might rely on this fact (this is similar to the hidden state in segment registers). A world switch would have the effect of spontaneously re-reading the PDPEs in the guest.

In case the VMM cares about emulating this effect, it can do so without additional h/w support, as follows. Every time the guest writes cr3 (or cr4?) while in (or entering) 32-bit PAE mode, the VMM can cache the four guest PDPEs. Any time the VMM constructs a shadow page table entry (which involves walking the guest page table), the VMM should use the PDPEs it has cached, rather than reading them from guest physical memory.

5.4.3 Protected Mode, Paging Disabled

When paging is disabled in the guest, the VMM virtualizes by physically running the guest with paging enabled. An additional intercept will be needed:

- `cr0` reads — since the physical copy of `cr0` has the PG bit set, the VMM must intercept reads of `cr0` and return the virtualized value expected by the guest.

The transition from paging enabled to disabled will be caught by the "baseline" intercept of cr0 writes.

5.4.4 Virtual x86 (v86) Mode

With the h/w support as defined, v86 mode should "just work" with the normal intercepts that the VMM would use for plain protected mode with paging enabled. Specifically, there are two areas of concern:

- Interrupt handling — when the VMM injects a virtual interrupt into the guest, that interrupt, from the point of view of the guest, is handled just like a physical interrupt. As long as v86 interrupt/exception intercepts take priority over any SVM intercepts, the "right" things will happen inside the guest (namely, a transition of the guest to protected mode).
- IOIO in v86 mode: as long as v86 IOIO intercepts take precedence over SVM-originated IOIO intercepts, an IN or OUT inside a v86 guest will transition the guest to non-v86 protected mode (in other words, to the "v86-VMM" inside the guest), just as would happen without

SVM.

As a result, v86 mode inside a guest should be totally transparent to the VMM, and be treated just like plain protected mode.

5.4.5 Real Mode

The VMM will place the guest in "pages real mode" (see Section 4.8), i.e., cr0.PE==0 but cr0..PG==1. Segments will work as expected in real mode, and since paging is still enabled, memory is protected — it's just that the shadow page table filler doesn't walk a guest page table before walking the host page table. All page faults are intercepted to the VMM, which does the "right thing" behind the guest's back. Reads of cr0 are intercepted, and the VMM supplies the value expected by the guest.

5.4.6 SMM

Basic SMM mode is essentially Real Mode with a few virtual interrupts masked (the VMM can track whether SMIs and NMIs are blocked in guest SMM by intercepting entry and exit to SMM as well as iret instructions. Entering and exiting SMM is somewhat complicated by the requirement that the VMM be able to run non-trusted platform SMM code inside a container. We devote the next section entirely to SMM handling.

5.5 Containerizing SMM

In this section, we describe in more detail how the VMM can achieve containerization of platform SMM code — the execution of "platform" SMM code inside a guest, in order to control the actions taken by the platform SMM code, and to control what parts of CPU, memory and device state are visible to it. Note that this is not a VM requirement, but rather a security requirement in cases where the VMM/hypervisor does not trust the platform SMM code, and hence must not execute it "natively (without the isolation afforded by a container).

We give two alternatives, corresponding to the two levels of proposed hardware support, see Section 4.10.4.1 and Section 4.10.4.2.

5.5.1 Basic Flow

This is the basic flow for "redirecting" an SMI into a container; the VMM is assumed to have set up a trusted SMM trampoline by changing SMM_BASE; the old value of SMM base has been noted in order to emulate the proper environment for the platform SMM code.

Bold lines marked "HW:" are hardware actions; the rest are s/w actions. (Let's for the moment defer the issue of internal versus external SMI sources, and the issue of SMIs that originate in the host rather than a guest.)

(running in guest, SMI arrives)
HW: vmexit (code SMI)

(back in hypervisor)
VMSAVE [we need all guest state]
full host state reload [prepare for interrupt]
record: source of SMI was guest, synchronous [so downstream s/w knows what to do]
gif := 1
HW: enter SMM [acknowledges SMI, SMMactive:=1, SMMSAVE]

(in hypervisor SMM trampoline)
turn on paging
set up SPT for SMM container: make visible the guest and the platform SMM code
emulate the guest's SMM-save:
    "copy" (most) guest VMCB state to container's SMsave area
    may need to clean state at this point
    set up I/O restart related information (from hypervisor's SMsave, or from MSRs)
emulate the guest's transition to SMM mode
    "copy" (most) guest VMCB state to container's VMCB, then...
    apply SMM register (eip:=smm_entrypt, CRs, etc...)
    but fake out CRs so container executes in paged-real mode
SVM (on container's VMCB)
HW: load container state from its VMCB
HW: execution starts at platform SMM entry point

(now in platform SMM code, SMM active, CRs "faked", paged real mode)
...(platform SMM code does its thing)...
    any intercept: save state back to VMCB, load host state (leave SMM on)
    host handles the intercept, switches back in here
...
RSM -> intercepted before it executes
HW: vmexit

(back in hypervisor trampoline, after the SVM in there)
(VMCB contains guest state before RSM; probably can discard)
diff container's SMsave area against what is was before platform SMM code ran
merge (some) diffs into guest VMCB or host state (via hypervisor's SMsave area)
[note: for interrupted *host* IOIO, may have to adjust the eip we RSM to below]
[TODO: hypervisor could apply some sanity checks, see below]
RSM [this one is not intercepted]
HW: exit SMM [signals SMI-end, SMMactive:=0, SMRESTORE]
[hypervisor may pick new guest to run]

Issues:

- Guest SMI caused by "INS" instruction: platform SMM code needs to write guest memory in order to emulate the INS. This means that we'd have to map the guest's address space read/write, and we'd lose (some) control over what the legacy SMM code accesses.
  The hypervisor could pretend to the platform SMM code that the SMI was caused by an "IN" instruction, and not allow the SMM to write any of the guest's memory. Then, if SMM indicates that it has emulated the I/O for us, the hypervisor can extract the return value from eax and store it in guest memory. (Yes, this essentially requires a disassembler / interpreter in the hypervisor. It's needed anyway.)
- Sanity checks: if the SMM indicates that the I/O instruction that caused the SMI should be

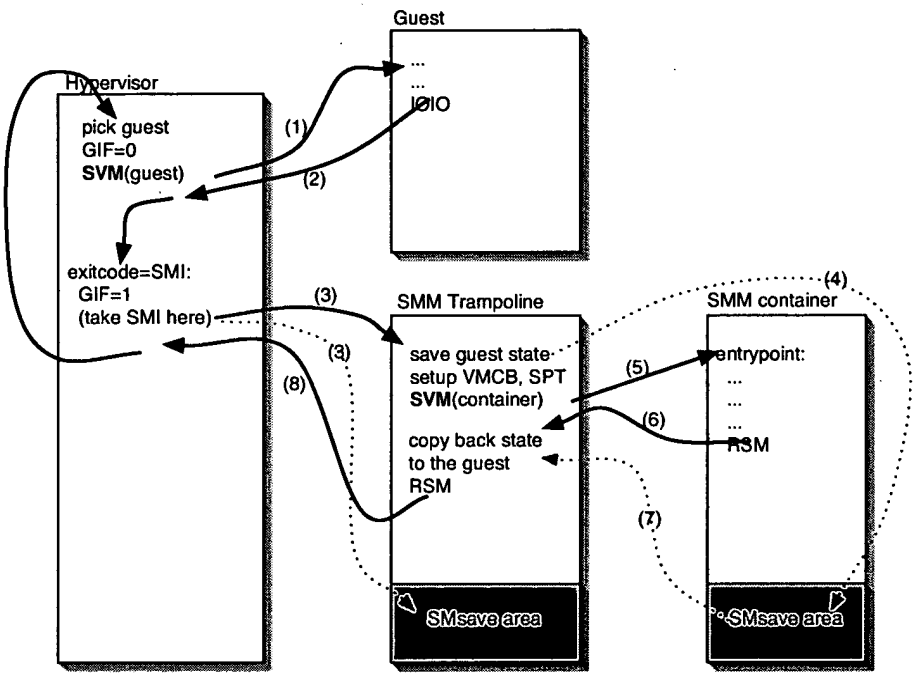
restarted, then it presumably hasn't emulated anything, and we should not copy back any registers from the container's SMsave area into the guest's VMCB.

- If the platform SMM code requests an I/O instruction restart, we presumably do not modify the guest state at all, but rather simply have it resume at the eip of the I/O instruction, in hopes that this time, it will go through.
- When we go to multiple guests, that raises the question of what memory space to make visible to the container which holds the platform SMM code. This is a policy issue for the VMM, and relates to what operations are permitted in SMM code; for example, this issue is complicated if the SMM code relies on any state to survive outside the SMRAM areas.

5.5.1.1 Basic Flow: SMI in Guest

This graphically shows the flow for control for an SMI that occurs while in the guest (possibly due to IOIO), and that needs to be handled by the platform SMM handler, inside a container.

Figure 6. Basic Flow, SMI In Guest



1. VMM runs the guest
2. Guest receives SMI, possibly due to IOIO
3. VMM enables GIF, so h/w takes the SMI and transitions to the VMM's SMM trampoline; the current VMM state is saved in the SMsave area.

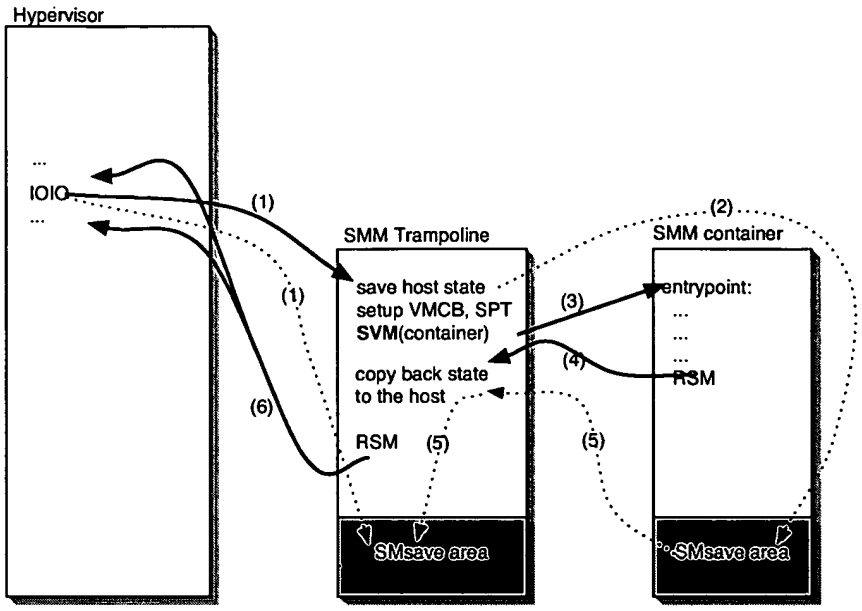
4. The trampoline "copies" guest state (suitably cleaned or munged) into the SMM container's SMsave area. To simulate the rest of the (virtualized) SMM transition, the guest state is also copied to a VMCB for the container and the register changes common to SMM are applied (e.g., eip := SMM container's entry point).
5. The VMM starts the container[1]. Any intercepts, e.g., to read CR0, return to the SVM, where the VMM handles them and resumes the container.
6. The final RSM in the container is intercepted (before it changes state).
7. Depending on whether an IOIO instruction was aborted by the original SMI, and depending on whether the platform SMM appears to have emulated the IOIO, some state from the container's SMsave area maybe copied back into the guest's VMCB; this may include an adjustment to the guest's eip. Note that the SMsave area in the trampoline remains unchanged (there's no need to change host state).
8. The trampoline's RSM completes (since it is not intercepted in the VMM), and takes control back to the VMM mainloop, which can now pick another guest to run.

5.5.1.2 Basic Flow: SMI in Host

Similar to SMI in guest, except that the trampoline (carefully!) may modify *host* state in its SMsave area before executing the RSM to return to the hypervisor.

1. The container is forced to run in "Paged Real Mode", which the VMM achieves by changing the paging-related CRs in the VMCB (but intercepting reads to the CRs, and "lying" about their true value).

Figure 7. Basic Flow, SMI in Host



1. Host receives SMI, possibly due to IOIO. (We assume GIF==1). H/w takes the SMI and transitions to the SMM trampoline; the current host state is saved in the SMsave area.
2. The trampoline "copies" host state (suitably cleaned or munged) into the SMM container's SMsave area. To simulate the rest of the (virtualized) SMM transition, the host state is also copied to a VMCB, where the register changes common to SMM are applied (e.g., eip := SMM container's entry point).
3. The trampoline code starts the container[1]. Any intercepts, e.g., to read CR0, return to the SVM, where the VMM handles them and resumes the container.
4. The final RSM in the container is intercepted (before it changes state).
5. Depending on whether an IOIO instruction was aborted by the original SMI, and depending on whether the platform SMM appears to have emulated the IOIO, some state from the container's SMsave area maybe copied back into the trampoline's SMsave area; this may include an adjustment to the host's eip (depending on whether it should retry any IOIO operation).
6. The trampoline's RSM completes (since it is not intercepted in the VMM), and takes control back to the VMM mainloop, which can resume execution.

1. The container is forced to run in "Paged Real Mode", which the VMM achieves by changing the paging-related CRs in the VMCB (but intercepting reads to the CRs, and "lying" about their true value).

5.5.2 Fast Flow

The basic flow is rather slow; about three times as slow as non-virtualized SMM. At least for synchronous internal SMIs inside guests, we can streamline (and simplify!) the flow, eliminating the SMM transitions.

This streamlined approach uses SVM's I/O permission bitmap to virtualize the I/O Trapping registers for guests; that way (a) we won't have to save/restore those registers when we switch guests, and (b) we get a faster IOIO intercept into the hypervisor, rather than a more heavyweight SMI.

On the other hand, if we use SVM's IOIO intercepts, then there won't be an SMI pending that magically whisks us into SMM mode as soon as we set GIF:=1 in the host; there would need to be a separate mechanism by which the host can manipulate SMMactive, and signal to external logic the assertion/deassertion of SMM. (Note that this can not be done via SVM, since we need to run some hypervisor code right after turning on SMM but before dispatching to a guest.)

The basic idea is as follows: when the hypervisor starts up, it examines the on-chip I/O Trapping registers. For any ports trapped this way, it notes (in a global s/w structure) that these ports require platform SMM intervention. The hypervisor then disables all I/O Trapping registers; there's no need to use them.

When a guest attempts to write the I/O Trapping registers, the hypervisor notes (in a per-guest s/w structure) that the guest wants to run its own SMM code in response.

In the SVM IOIO bitmap for a guest, the hypervisor only leaves accessible IOIO ports that neither guest nor platform SMM nor hypervisor is "interested" in. When the hypervisor gets an IOIO trap, it consults the global and per-guest tables to determine who should handle the synchronous SMI: guest SMM, platform SMM, or hypervisor itself.

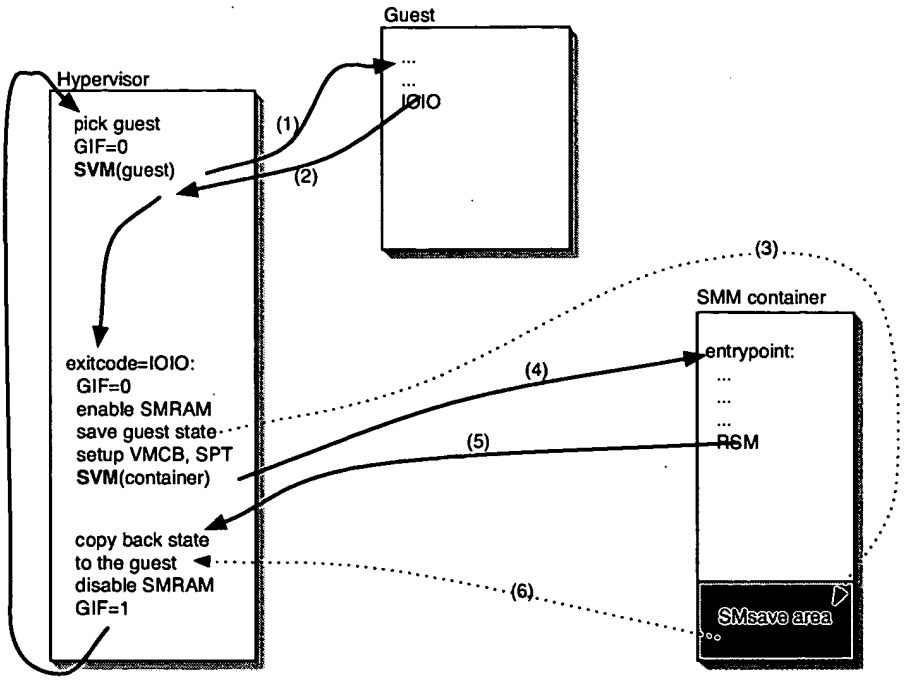
```
(running in guest, IN/OUT[S] triggers IOIO intercept, which was set up to virtualize internal synchronous SMI)
HW: vmexit (code IOIO; IOIO uCode has saved eip,ecx,edi,esi,size, ...)

(back in hypervisor)
VMSAVE [we need all guest state]
full host state reload [worst case, can probably do better]
analyze IOIO info (in MSRs?); for this example we assume we need platform SMM
record: source of SMI was guest, synchronous [so downstream s/w knows what to do]

** cause h/w to acknowledge SMI (?), enable SMRAM (but no time-consuming SMMSAVE)
** this must also (atomically) block SMIs (we could use GIF...)

set up SPT for SMM container: make visible the guest and the platform SMM code
emulate the guest's SMM-save:
   "copy" (most) guest VMCB state to container's SMsave area
   may need to clean state at this point
   set up I/O restart related information (from hypervisor's SMsave, or from MSRs)
emulate the guest's transition to SMM mode
   "copy" (most) guest VMCB state to container's VMCB, then...
```

```
   apply SMM register (eip:=smm_entrypt, CRs, etc...)
   but fake out CRs so container executes in paged-real mode
SVM (on container's VMCB)
HW: load container state from its VMCB
HW: execution starts at platform SMM entry point

(now in platform SMM code, SMM active, CRs "faked", paged real mode)
...(platform SMM code does its thing)...
   any intercept: save state back to VMCB, load host state (leave SMM on)
   host handles the intercept, switches back in here
...
RSM -> intercepted before it executes
HW: vmexit

(back in hypervisor, after the SVM that launched the container in the first place)
(VMCB contains guest state before container's RSM; probably can discard)
diff container's SMsave area against what is was before platform SMM code ran
merge (some) diffs into guest VMCB

** cause h/w to signal SMI-end (?) & set SMMactive:=0; no SMRESTORE
** unblock SMIs
[hypervisor may pick new guest to run]
```

Figure 8 below shows this flow graphically. Note that there is no longer a need for a transition to a trusted SMM trampoline code; all "wrapper" work is done directly inside the VMM, including the activation or deactivation of SMM (ASeg, TSeg regions; interrupt blocking), and external assertion or deassertion of the SMI-active signal.

IMPORTANT: activation/deactivation of SMM cannot be done as part of the SVM instruction, since the VMM needs visibility into the SMM container (i.e., SMM active) even when the guest is not running — e.g., to set up the SMsave area, and to handle intercepts from the container.

Figure 8. Fast Flow: Virtual SMI in Guest



5.5.3 Different SMI Causes

When using one of the above flows to enter containerized SMM, the SMM needs to present the SMM code with an environment depending on what caused the SMI; likewise, when the containerized SMM exits, the hypervisor needs to decide how to use the state left by the SMM code.

Any time the hypervisor invokes (explicitly or implicitly) platform SMM, it needs to know

- was an IOIO aborted by SMM?
- if so, was that in host or guest?

Where do we get all this info?

- all IOIO instructions (whether in host or guest) save relevant state from before the IOIO insn, and (I am assuming!) a flag that indicates whether the IOIO insn was "aborted" by an internal or external SMI.[1]
- when exiting a guest, the hypervisor can note that (a) the SMI was received inside a guest, and (b) extract the above IOIO information before anyone executes another IOIO
- when a guest exits with an IOIO intercept, and the hypervisor decides to invoke the platform

1. If the IOIO insn occurred in host code, then the IOIO information is not only available in the hidden registers written by IO/OUT uCode, but also (well, most of it) in the hypervisor's SMsave area after it has taken the SMI.

SMM on its behalf, again we know that this is the result of guest IOIO.

- (note that physical SMM transitions always occur inside hypervisor code, after the hypervisor sets GIF=1; this is true regardless of whether host or guest "caused" or observed the SMI)
- when the guest exited for any other reason, any SMI must have been caused by, or seen by, host code, and (presumably) no interaction with any guest is required.

On exit from platform SMM (which indicates in its SMsave area whether it wants an IOIO instruction retried), s/w should handle as follows:

Table 16: S/w action after platform SMM has finished

IOIO aborted?	in host or guest?	SMM requests IO restart?	action
yes	host	yes	resume running host at the IOIO insn where the SMI interrupted; the eip (+/- restart adjustment) is in hypervisor's SMsave area
		no	for IN insn, copy eax from platform SMMsave area to host's eax, resume host at the insn *after* the interrupts IOIO; the eip (+/- restart adjustment) is in hypervisor's SMsave area
	guest	yes	no change to guest state; next time it's scheduled, it should resume running at the IOIO insn where it was interrupted
		no	for IN insn, copy eax from platform SMMsave area to guest's eax (in VMCB); next time guest is scheduled, it should resume *after* the interrupted IOIO insn.
no	n/a	must be "no"	no change to host state, just resume running (unless the hypervisor needs to be informed about external events, too, in which case we need cooperation between platform SMM and hypervisor)

5.6 IOIO

The VMM can choose which I/O ports to make available to a guest, by setting the bits in the VMCB's I/O permissions map (IOPM) accordingly. The IOPM can also be used to virtualize the CPU-internal I/O trapping mechanism, see Section 5.9.

5.7 MMIO

For para-virtualized guests that have direct access to I/O devices, the device's control registers would be mapped into the shadow page table by the VMM, giving the guest direct access. For virtual devices, MMIO device registers have to be emulated: the VMM needs to map the respective region as not present in the shadow page table, so a guest's access to the virtual device causes a page fault. The

VMM needs to analyze the information on the page fault, recognize that it represents an access to a virtual device, and emulate the access in software.

5.8 A20

Virtualize by trapping the IOIO accesses that change A20, so the VMM can track the current state of the guest's (virtual) A20 pin. The VMM needs to factor in that value when updating the shadow page table. Note that typically, the VMM needs to keep physical A20 masking turned off at all times.

5.9 I/O Trapping

The K8 has several MSRs through which is can intercept IN/OUT instructions to selected ports (or PCI Config Space) and invoke SMM handlers. The VMM can emulate these registers by intercepting guest access to these MSRs (through the MSR intercept mechanism), recording which ports are being protected by the guest, and placing a corresponding protection in the I/O permission bitmap for that guest. This way, the physical I/O Trapping registers can be reserved entirely for the VMM's private use. Also, since their function has been folded into the IOPM, the VMM need not save/restore then when switching to a different guest.

5.10 Interrupts

5.10.1 INTR and LocalAPIC

When the currently running guest *owns* the INTR interrupts, any such interrupts are handled exactly the same as in host mode: a physical INTR (when enabled according to rflags.if and the interrupt shadow) causes an INTAK cycle to the localAPIC, and a dispatch to the appropriate interrupt handler. Likewise, the guest may interact directly with the localAPIC, and mask/unmask (in rflags) physical INTR interrupts.

If the guest does not own INTR (i.e., the INTR intercept bit is "1"), then the hardware support described in Section 4.9 ensures that the CPU-internal interrupt masking functions are virtualized without VMM intervention, and that both physical interrupt delivery and all localAPIC interactions other than changing TPR are intercepted to the VMM. This allows the VMM to maintain a software model of a localAPIC for each guest.

Interrupt delivery follows this scenario:

- a physical INTR interrupt arrives; if a guest is currently running, the guest exits to the VMM
- the VMM takes the interrupt (as soon as GIF=1 after the world switch), and immediately masks it in the IMR
- the VMM examines the vector number and determines the guest for which the interrupt is des-

tined

- the VMM updates a s/w model of the guest's localAPIC (in the process probably translating the physical vector number to a vector in the guest).
- from the guest's localAPIC model, the VMM can determine the priority and vector of the highest-priority interrupt for that guest; it delivers that interrupt to the guest by setting the `v_irq`, `v_intr_prio` and `v_intr_vector` fields in the guest's VMCB. The next time the guest runs, it will take the interrupt (as soon as it is enabled in `eflags.if` and the virtualized `v_tpr` register).

At some point later, a well-behaved guest will signal the end of interrupt processing by issuing an EOI (end-of-interrupt) to the localAPIC. The VMM must intercept this operation, so that

- the VMM translates the guest's interrupt number in the EOI to the physical interrupt number,
- the VMM unmasks the interrupt in the IMR and issues an EOI to the physical localAPIC on behalf of the guest.

TODO: this requires two world switches per interrupt delivered and processed— will the resulting performance be acceptable?

5.10.2 Blocking INTR While Running Guest

As defined, the arrival of a physical INTR (of priority > APIC.TPR, while phys_if==1) causes an exit from guest mode (see Figure 3 on page 39). If the VMM desires to block physical interrupts while executing guest code, it can clear `phys_if` or set the localAPIC's TPR to the maximum value. Presumably, when doing so, the VMM will periodically reenable interrupts in order to poll for pending interrupts.

5.10.3 SMI

Note: because of the way we virtualize SMM (the VMM needs to be aware of when the guest enters and exits SMM), SMI interrupts must always go through the VMM. If the VMM needs to inject an SMI into the guest, it needs to simulate the entire SMM-entry and exit sequence in software (and place the guest in Paged Real Mode). This means that the VMM also needs to track RSMs, and whether IRET has reenabled NMIs.

5.10.4 NMI, INIT

NMI and INIT, when intercepted, cause an exit from the guest; additionally, the VMM can redirect a host-level INTR into an #SX exception. To simulate either in a guest, the VMM must emulate the entire operation in software. For a virtual INIT, the VMM would reset the guest's state in the VMCB (and any an ciliary s/w structure the VMM may use) in accordance with the semantics of an INIT. For a virtual NMI, the VMM needs to locate the proper entry in the guest's IDT, and simulate the

entire interrupt dispatch sequence. After having delivered a virtual NMI to a guest, the VMM needs to intercept the next guest IRET, so the VMM can track whether NMIs are currently masked in the guest.

5.10.5 RESET

Reset cannot be intercepted. However, the VMM can "reset" a guest by appropriately changing the guest's state in the VMCB (and any device models, cached guest state, etc.).

5.11 Machine Check, HALT, Shutdown

These events will always cause an exit to the VMM, which can then process them (and potentially inject virtualized equivalents into a guest).

5.12 Timers and Performance Counters

The VMM can intercept guest access to the timestamp counter; this certainly allows us to emulate various policies for virtualization. For better performance, and if hardware supports it, the VMM can specify an offset to be added to reads of the TSC; this should capture some common usage patterns efficiently (i.e., without intercept to the VMM).

With the proposed h/w support, the only option the VMM has for virtualizing performance counters is to intercept all accesses to performance counters.

5.13 CPUID

Virtualizing CPUID allows the VMM to creatively "lie" to the guest about the identity of the virtual machine. Again, the simplest solution would be to intercept all uses of CPUID. However, some uses of CPUID are assumed to be fast; a trap to the VMM may not be acceptable in those cases. Additional h/w support may be desirable.

5.14 Devices

We want the VMM to be able to control which devices are visible to the guest. To that end, the VMM should intercept all accesses to PCI config space (this is handled by intercepting the corresponding IOIO operations), and present to the guest the real and/or physical devices that the guest is supposed to see.

5.14.1 Virtual Devices

To create a virtual device for a guest, the VMM makes the device visible in the guest's PCI config space. When the guest allocates MMIO and IOIO space for the device (via the PCI config mechanism), the VMM must intercept this setup and associated addresses with the virtual device. The VMM will trap all accesses to those address ranges and perform a software simulation of the virtual device. Note that virtual devices may accurately reflect real physical devices, or they may implement some abstract device that has presumably been streamlined for more efficient operation in a VM environment (in which case the guest would need matching device drivers).

The big drawbacks to virtual devices are

- the effort to write a model for each (new) device one is interested in
- the need for custom driver for "abstract" device models
- performance

5.14.2 Direct Device Access

The VMM grants a guest direct access to devices by not only making them visible to the guest in the PCI config space, but actually mapping their MMIO control registers into the guest's address space. To protect other guests from DMA by these devices, the VMM must set up appropriate protection domains (each essentially described by a DEV table) in the Northbridge DEV registers, and set up appropriate mappings from device IDs to protection domains.

Note that the latter mapping currently relies on HT bus and device IDs, which somewhat limits the granularity at which ownership of devices can be assigned. In the future, we expect to rely on PciExpress deviceIDs. However, this will in all likelihood require a rather different structure for mapping device IDs to domains; hence this portion of VMM setup code will be platform specific.

When the VMM exposes a device for direct access by a guest, it must do the following

- the VMM must know the PCI bus and device number, so it can intercept PCI config space accesses;
- the VMM must know the deviceID (i.e., the HT bus ID) for the device;
- the VMM must set up a protection domain (i.e., a DEV table) in the Northbridge that specifies what memory the device may access. Typically (but not always), this will be the physical address space of the guest;
- the VMM must ensure that the deviceID is mapped to the correct protection domain in the Northbridge;
- the VMM must track PCI config accesses to the device, so it can make the device visible, and to track guest assignments of MMIO/IOIO space and interrupt vector(s). The former are required so the VMM can add the proper mappings when it fills in the guest's shadow page tables.
- The exact details of how the VMM virtualizes guest-assigned interrupt vectors are *TODO*, but

essentially, the VMM needs to be able to infer the interrupt vector that the platform will deliver for a given setup of the PCI config registers (this is platform dependent), so that it may (a) infer the final interrupt vector the guest expects to show up at the localAPIC[1], and (b) construct a config-register setting for the device that will produce an interrupt vector that is still available at the real ioAPIC and localAPIC.

5.14.3 Reassigning Device Ownership

Ideally, the VMM should be able to reassign devices to different owners (i.e., guests). If this is to be done without rebooting the "donor" and "receiver" guests, some cooperation from the guests is needed (para-virtualization).

Moreover, the device being reassigned should be put into a quiescent state. However, there is no standardized way of resetting a device (and the hypervisor should definitely not be burdened with drivers for all possible devices). The following, slightly more cumbersome, scheme should work

- via negotiation with the VMM, the donor guest agrees to relinquish the device
- since the donor guest contains a device driver, it is able to quiesce the device
- the VMM places the device in a domain that has no DMA access rights at all
- the VMM hands the device to the receiving guest
- since the receiving guest contains a device driver, it is able to re-initialize the device
- after the device has been re-initialized (signal from receiver to VMM), the VMM places the device in a domain that gives access to the receiver's address space.

This works in a non-trusted environment, since

- if the donor fails to quiesce the device, its secrets may be exposed to the receiver, but it cannot access or damage the receiver's memory via outstanding device operations
- if the receiver fails to re-initialize the device, it may expose itself to the donor, but it cannot access or damage the donor.

5.14.4 Issue with Late Boot

SEM allowed a "late boot" of the secure kernel via the SKINIT instruction (the SK/hypervisor would be "booted" after a regular operating system, e.g., Windows, had already booted). There are potential problems with this, concerning direct access to devices. Typically, once the initial O/S is up and running, it will have enumerated, found and configured all physical devices in the system. At this point, it is difficult for the hypervisor to "take away" any of the devices, in order to assign them to a guest. One solution would be to boot the hypervisor first. Another would be to provide the necessary level of cooperation between the initial O/S and the hypervisor.

1. This will require the VMM to track how the guests program their (virtualized) ioAPIC.

5.15 Guest Runtime Limit

The VMM will need to limit how long any guests can run; typically it might do that by setting up a timer interrupt. More useful, especially for debugging uses, would be an option in the VMCB to run a guest for an *exact* number of instruction retired. It is unlikely that we will be able to provide such support in the first incarnation of VM support, so the VMM will have to rely on timer interrupts or the like for limiting how long guests can run.

5.16 Virtualizing Control Registers

In this section, we describe how to virtualize the various functions in the core control registers.

5.16.1 rflags

Accessed by: pushf, popf, sti, cli, iret, task switch

- IF bit — can run native in the guest, thanks to interrupt h/w support.
- VM bit — can run native in the guest, since h/w handles interrupts and v86-related exceptions.
- all other bits — trivially run native in guest (and are saved/restore across world switches).

5.16.2 cr0

Accessed by: mov to/from cr, lmsw, smsw, clts

- PE — use natively in the guest
- MP — use natively in guest
- EM — use natively in guest
- TS — use natively in guest
- NE — if cleared, VMM must emulate FERR# and IGNNE.
- WP — must intercept writes, since this affects how shadow page tables are built
- AM — use natively in guest
- NW,CD — these affect the memory types used in shadow page tables; must intercept
- PG — must intercept, since this affects shadow page table formation, and real mode emulation.

Typically, the VMM will not have to intercept reads — except possibly in real mode, if we should emulate real mode by running with PE==0, PG==1 (currently an illegal mode); the VMM has to intercept cr0 reads in order to substitute the virtualized value.

5.16.3 cr2

Accessed by: mov to/from cr, page fault

Must intercept writes (other than page fault) to this register, so that the VMM can track the true value visible to the guest; that value may get overwritten by page faults that are invisible to the guest[1] (and therefore should not in fact appear to change cr2). No need to intercept reads.

5.16.4 cr3

Accessed by: mov to/from cr, task switch.

With shadow page tables are used, the physical cr3 points at the shadow page table. Therefore, the VMM must intercept reads (to deliver the virtualized value, i.e., the pointer to the guest's page table) and writes (so the VMM can update its shadow page tables).

Note: this is written by task switches as well. It may be best to simply intercept all task switches, and emulate them entirely in the VMM.

5.16.5 cr4

Accessed by: move to/from cr.

- VME, PVI — use natively in guest.
- TSD — use natively in the guest.
- DE — use natively in the guest.
- PSE, PAE — must intercept writes, since these bits affect shadow page table formation
- MCE — use natively in the guest, but likely intercept #MC so host can attempt to contain faults to a given guest.
- PGE — intercept writes, to VMM can invalidate shadow page tables.
- PCE — use natively. The VMM can request explicit intercepts of rdpmc.
- OSFXSR — use natively.
- OSXMMEXCPT — use natively.

Overall, writes to bits that can be used natively in the guest are probably rare, so the VMM might just as well intercept *all* writes to this register. If the VMM ever builds a shadow page table that uses a different "mode" than the guest[2], the VMM will also have to intercept reads.

1. For example, a page fault taken because an entry in the shadow page table has not yet been filled in — yet the translation is valid in both the guest and host page tables. The VMM will restore the guest's cr2, and resume execution after filling in the shadow page table.

5.16.6 cr8

The localAPIC's TPR (available as `cr8` in AMD long mode) is written very frequently in Windows operating systems. Therefore, we should provide h/w support that allows guests to write this register without VMM intervention (yet only control virtual interrupts, not physical ones).

5.17 Virtualizing Selected MSRs

By default, we envision that access to all MSRs be intercepted, and the MSRs virtualized in software. For now, we discuss how to virtualize a few selected MSRs.

5.17.1 apic_base

The VMM needs to intercept writes to this register, so it can track when the guest attempts to access its localAPIC. The VMM could trap APIC accesses by marking as "not present" in the shadow page table entries that map to this guest physical address.

Since reads of this register are likely very rare, the VMM should intercept reads as well, at which point it can virtualize this register entirely in software; it will not need to be saved/restored across world switches.

5.17.2 efer

Save/restore on world switch (unless host and guest values just happen to match). However, *also* intercept all writes so the VMM can track the guest entering and exiting long mode.

5.17.3 pat

Intercept all writes, and possibly all reads as well. If the host `pat` is set up just right and provides at least all memory types available in the guest's (virtual) `pat`, all translation of guest to host memory types can be done by the VMM when constructing the shadow page table. It is also of course always possible to world-switch the `pat` register, should that prove necessary.

5.17.4 star, lstar, cstar, sfmask

If the guest is to be able to use these registers natively, the VMM must save/restore these registers for the guest. IMPLEMENTATION DETAIL: if there is demand, we can make this available as an option on VMLOAD/VMSAVE.

2. Consider a guest that uses plain legacy paging, while the VMM runs in long mode — *and* uses physical addresses above 4GB to hold guest pages. The VMM would have to use PAE or long-mode paging to build the shadow page table, meaning the value in the physical cr4 would be different from the guest's value.

5.17.5 sysenter_cs, sysenter_esp, sysenter_eip

If the guest is to be able to use these registers natively, the VMM must save/restore these registers for the guest. IMPLEMENTATION DETAIL: if there is demand, we can make this available as an option on VMLOAD/VMSAVE.

5.17.6 KernelGSbase

Handle natively, no intercept. The VMM should save/restore as needed. Note: if the guest is not in long mode (and we intercept all transitions to/from long mode) then the VMM can omit the save/restore. Likewise if the host doesn't use long mode, save/restore is only needed when switching to another guest that might use this register. Therefore, save/restore should either be done in s/w, or there should be an option in the VMCB — this must not be a compulsory save/restore.

5.17.7 gs_base and fs_base

Handle natively, no intercept.

5.18 Virtualizing Various Privileged Instructions

5.18.1 cpuid

Intercept and emulate.

5.18.2 mov to/from cr, lmsw, smsw

Intercept and emulate writes. Some execution modes may require interception of reads. H/w may add options to intercept only writes that change selected "sensitive" bits.

5.18.3 clts

As long as only selective cr0 is active (the preferred mode of operation), runs natively in the guest.

5.18.4 pushf, popf, cli, sti

With the proposed h/w support for maskable interrupts, these instructions will all run natively. (Recall: writes to eflags.if will only write-through to the phys_if bit if the processor currently "owns" maskable interrupts; phys_if controls physical interrupts.

5.18.5 mov to/from dr

This intercept is not required for normal VMM operation, but may be useful if the VMM wants to inspect or mediate debug register usage by a guest. Otherwise, the debug register should work natively in the guest. The VMM will want to either (a) swap all DRs on world switch, or at least (b) set up the host's dr7 to "all DRs disabled" before starting the guest, and requesting that SVM save/restore dr7 at least. That way, debug register settings from the guest cannot trigger inside VMM code.

5.18.6 rdmsr/wrmsr

Give access as necessary in the MSR permission table; intercept and emulate others.

5.18.7 rdpmc

Intercept and emulate.

5.18.8 rdtsc

Intercept and emulate, or (if supported by h/w) provide offset relative to physical tsc.

5.18.9 lgdt, lidt, sidt, sgdt

This intercept is not required for normal VMM operation, but may be useful if the VMM wants to inspect or mediate usage of the associated resources by a guest.

5.18.10 lldt, ltr, sldt, str

This intercept is not required for normal VMM operation, but may be useful if the VMM wants to inspect or mediate usage of the associated resources by a guest.

5.18.11 iret

In some guest modes of operation, iret may need to be intercepted in SMM, so VMM can track whether interrupts are enabled.

5.18.12 hlt

Typically intercepted. VMM can switch to another guest.

5.18.13 int*x*

Use natively in the guest.

5.18.14 invd/wbinvd

Intercept the former; most likely treat it as a no-op. The latter is safe.

5.18.15 invlpg

VMM may or may not intercept, depending on how guest physical addresses are remapped.

5.18.16 rsm

Intercept and emulate entirely in software. Note that SVM/VMLOAD/VMSAVE provide the necessary access to all hidden state. The SVM intercept is checked before any exception checks, so the VMM needs to perform those itself.

5.18.17 syscall/sysret

Will work natively (i.e., without VMM intervention) in the guest; the VMM will have to save/restore the associated registers (star, cstar, lstar, sfmask) if it needs to use them itself, or if it is about to switch to running another guest.

5.18.18 sysenter/sysexit

Will work natively (i.e., without VMM intervention) in the guest; the VMM will have to save/restore the associated registers (sysenter_cs, sysenter_esp, sysenter_eip) if it needs to use them itself, or if it is about to switch to running another guest.

5.18.19 swapgs

Use natively in the guest.

5.19 Need for Interpreter

The VMM needs to emulate operations that access physical memory ranges that are not mapped into the guest's address space, either because they represent MMIO ranges for virtual devices (including a localAPIC model), or because there is no memory or device present at the address in question.

In practice, this will require that the VMM include a complete x86 interpreter, unless significant hardware support is added. Consider a slightly contrived example: a guest executing an `fadd` operation that takes its source from memory, when that memory region happens to map to the (virtual) localAPIC's MMIO range. The guest memory access will cause a page fault, and the actual raw read of the localAPIC registers has to be emulate by VMM software. This means that the VMM must be able to recognize the instruction that faulted, merely to infer how many bytes to read from what address. Moreover, since there is no way to "inject" the raw data read back into the guest, the VMM must also emulate the operation of the fadd instruction itself.

This not only raises the complexity barrier on the VMM by quite a bit, it also creates problems for backwards compatibility — it would be nice if at least "harmless" (typically non-privileged) extensions in future processors just worked without changes to the VMM (specifically, without requiring the new instruction or behavior to be added to the VMM's interpreter).

Addendum

The Secure VM Support functionality described in the preceding section may be implemented within a variety of specific computer system architectures, as desired. For example, Fig. 9 is a block diagram of one embodiment of a computer system 200 that may implement the secure VM support functionality described above including processor 10 coupled to a variety of system components through a bus bridge 202 is shown. In the depicted system, a main memory 204 is coupled to bus bridge 202 through a memory bus 206, and a graphics controller 208 is coupled to bus bridge 202 through an AGP bus 210. Finally, a plurality of PCI devices 212A-212B are coupled to bus bridge 202 through a PCI bus 214. A secondary bus bridge 216 may further be provided to accommodate an electrical interface to one or more EISA or ISA devices 218 through an EISA/ISA bus 220. Processor 10 is coupled to bus bridge 202 through a CPU bus 224 and to an optional L2 cache 228. Together, CPU bus 224 and the interface to L2 cache 228 may comprise an external interface to which external interface unit 18 may couple. The processor 10 may include the processor-specific secure VM support functionality as described in the preceding section.

Bus bridge 202 provides an interface between processor 10, main memory 204, graphics controller 208, and devices attached to PCI bus 214. When an operation is received from one of the devices connected to bus bridge 202, bus bridge 202 identifies the target of the operation (e.g. a particular device or, in the case of PCI bus 214, that the target is on PCI bus 214). Bus bridge 202 routes the operation to the targeted device. Bus bridge 202 generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus 214, secondary bus bridge 216 may further incorporate additional functionality, as desired. An input/output controller (not shown), either external from or integrated with secondary bus bridge 216, may also be included within computer system 200 to provide operational support for a keyboard and mouse 222 and for various serial and parallel ports, as

desired. An external cache unit (not shown) may further be coupled to CPU bus 224 between processor 10 and bus bridge 202 in other embodiments. Alternatively, the external cache may be coupled to bus bridge 202 and cache control logic for the external cache may be integrated into bus bridge 202. L2 cache 228 is further shown in a backside configuration to processor 10. It is noted that L2 cache 228 may be separate from processor 10, integrated into a cartridge (e.g. slot 1 or slot A) with processor 10, or even integrated onto a semiconductor substrate with processor 10.

Main memory 204 is a memory in which application programs are stored and from which processor 10 primarily executes. A suitable main memory 204 comprises DRAM (Dynamic Random Access Memory). For example, a plurality of banks of SDRAM (Synchronous DRAM), double data rate (DDR) SDRAM, or Rambus DRAM (RDRAM) may be suitable. Main memory 204 may include the system memory 42 shown in Fig. 1. In one embodiment, main memory 204 may store code executable by processor 10 to implement the secure VM support functionality as described in the preceding section.

PCI devices 212A-212B are illustrative of a variety of peripheral devices. The peripheral devices may include devices for communicating with another computer system to which the devices may be coupled (e.g. network interface cards, modems, etc.). Additionally, peripheral devices may include other devices, such as, for example, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device 218 is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller 208 is provided to control the rendering of text and images on a display 226. Graphics controller 208 may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory 204. Graphics controller 208 may therefore be a master of AGP bus 210 in that it can request and receive access to a target

interface within bus bridge 202 to thereby obtain access to main memory 204. A dedicated graphics bus accommodates rapid retrieval of data from main memory 204. For certain operations, graphics controller 208 may further be configured to generate PCI protocol transactions on AGP bus 210. The AGP interface of bus bridge 202 may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display 226 is any electronic display upon which an image or text can be presented. A suitable display 226 includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system 200 may be a multiprocessing computer system including additional processors (e.g. processor 10a shown as an optional component of computer system 200). Processor 10a may be similar to processor 10. More particularly, processor 10a may be an identical copy of processor 10. Processor 10a may be connected to bus bridge 202 via an independent bus (as shown in Fig. 15) or may share CPU bus 224 with processor 10. Furthermore, processor 10a may be coupled to an optional L2 cache 228a similar to L2 cache 228.

Turning now to Fig. 10, another embodiment of a computer system 300 is shown. In the embodiment of Fig. 10, computer system 300 includes several processing nodes 312A, 312B, 312C, and 312D that may implement the secure VM support functionality described above. Each processing node is coupled to a respective memory 314A-314D via a memory controller 316A-316D included within each respective processing node 312A-312D. Additionally, processing nodes 312A-312D include interface logic used to communicate between the processing nodes 312A-312D. For example, processing node 312A includes interface logic 318A for communicating with processing node 312B, interface logic 318B for communicating with processing node 312C, and a third interface logic 318C for communicating with yet another processing node (not shown). Similarly, processing node 312B includes interface logic 318D, 318E, and 318F; processing node 312C includes interface logic 318G, 318H, and 318I; and processing node 312D includes

interface logic 318J, 318K, and 318L. Processing node 312D is coupled to communicate with a plurality of input/output devices (e.g. devices 320A-320B in a daisy chain configuration) via interface logic 318L. Other processing nodes may communicate with other I/O devices in a similar fashion.

Processing nodes 312A-312D implement a packet-based link for inter-processing node communication. In the present embodiment, the link is implemented as sets of unidirectional lines (e.g. lines 324A are used to transmit packets from processing node 312A to processing node 312B and lines 324B are used to transmit packets from processing node 312B to processing node 312A). Other sets of lines 324C-324H are used to transmit packets between other processing nodes as illustrated in Fig. 10. Generally, each set of lines 324 may include one or more data lines, one or more clock lines corresponding to the data lines, and one or more control lines indicating the type of packet being conveyed. The link may be operated in a cache coherent fashion for communication between processing nodes or in a noncoherent fashion for communication between a processing node and an I/O device (or a bus bridge to an I/O bus of conventional construction such as the PCI bus or ISA bus). Furthermore, the link may be operated in a non-coherent fashion using a daisy-chain structure between I/O devices as shown. It is noted that a packet to be transmitted from one processing node to another may pass through one or more intermediate nodes. For example, a packet transmitted by processing node 312A to processing node 312D may pass through either processing node 312B or processing node 312C as shown in Fig. 10. Any suitable routing algorithm may be used. Other embodiments of computer system 300 may include more or fewer processing nodes then the embodiment shown in Fig. 10.

Generally, the packets may be transmitted as one or more bit times on the lines 324 between nodes. A bit time may be the rising or falling edge of the clock signal on the corresponding clock lines. The packets may include command packets for initiating transactions, probe packets for maintaining cache coherency, and response packets from responding to probes and commands.

Processing nodes 312A-312D, in addition to a memory controller and interface logic, may include one or more processors. Broadly speaking, a processing node comprises at least one processor and may optionally include a memory controller for communicating with a memory and other logic as desired. Each processor may include the processor-specific secure VM support functionality as described in the preceding section.

Memories 314A-314D may comprise any suitable memory devices. For example, a memory 314A-314D may comprise one or more RAMBUS DRAMs (RDRAMs), synchronous DRAMs (SDRAMs), DDR SDRAM, static RAM, etc. The address space of computer system 300 is divided among memories 314A-314D. Each processing node 312A-312D may include a memory map used to determine which addresses are mapped to which memories 314A-314D, and hence to which processing node 312A-312D a memory request for a particular address should be routed. In one embodiment, the coherency point for an address within computer system 300 is the memory controller 316A-316D coupled to the memory storing bytes corresponding to the address. In other words, the memory controller 316A-316D is responsible for ensuring that each memory access to the corresponding memory 314A-314D occurs in a cache coherent fashion. Memory controllers 316A-316D may comprise control circuitry for interfacing to memories 314A-314D. Additionally, memory controllers 316A-316D may include request queues for queuing memory requests. Memories 314A-314D may store code executable by the processors to implement the secure VM support functionality as described in the preceding section

Generally, interface logic 318A-318L may comprise a variety of buffers for receiving packets from the link and for buffering packets to be transmitted upon the link. Computer system 300 may employ any suitable flow control mechanism for transmitting packets. For example, in one embodiment, each interface logic 318 stores a count of the number of each type of buffer within the receiver at the other end of the link to which that interface logic is connected. The interface logic does not transmit a packet unless the receiving interface logic has a free buffer to store the packet. As a receiving buffer is

freed by routing a packet onward, the receiving interface logic transmits a message to the sending interface logic to indicate that the buffer has been freed. Such a mechanism may be referred to as a "coupon-based" system.

I/O devices 320A-320B may be any suitable I/O devices. For example, I/O devices 320A-320B may include devices for communicating with another computer system to which the devices may be coupled (e.g. network interface cards or modems). Furthermore, I/O devices 320A-320B may include video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards, sound cards, and a variety of data acquisition cards such as GPIB or field bus interface cards. It is noted that the term "I/O device" and the term "peripheral device" are intended to be synonymous herein.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated.