Hardware Assisted Virtualization
Intel Virtualization Technology

Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

Buenos Aires, Junio de 2008

Index

1 Background, motivation and introduction to Intel Virtualization Extensions

1.1 Challenges to virtualizing Intel architecture

Established and emerging applications motivate strong support for virtualization in both server and client computing systems. Unfortunately, the IA-32 and Itanium architectures impose many challenges to providing such support. Software techniques exist that address some of those challenges.

Intel microprocessors provide protection based on the concept of a 2-bit privilege level, using 0 for most-privileged software and 3 for the least privileged. The privilege level determines whether privileged instructions, which control basic CPU functionality, can execute without fault; it also controls address-space accessibility based on the configuration of the processor's page tables and, for IA-32, segment registers. Most IA software uses only privilege levels 0 and 3, as Figure 1a illustrates. For an OS to control the CPU, some of its components must run with privilege level 0. Because a VMM cannot allow a guest OS such control, a guest OS cannot execute at privilege level 0. Thus, IA-based VMMs must use ring deprivileging, a technique that runs all guest software at a privilege level greater than 0. A VM could deprivilege a guest OS by running it either at privilege level 1 (the 0/1/3 model) or at privilege level 3 (the 0/3/3 model).

Figures 1b and 1c illustrate these choices. Although the 0/1/3 model supports simpler VMMs, it cannot be used on IA-32 processors for guests in 64-bit mode. The 64-bit mode is part of Intel's EM64T (Extended Memory 64 Technology), the 64-bit extension to IA-32. Ring deprivileging causes numerous virtualization challenges. Intel virtual technology extensions (vt-x) solve virtualization challenges in part by allowing guest software to run at its intended privilege level. Guest software is constrained, not by privilege level, but because —for VT-x— it runs in VMX non-root operation. Figure 1d illustrates this usage.

1.1.1 Ring aliasing

Ring aliasing refers to problems that arise when software is run at a privilege level other than the level for which it was written. An example in IA-32 is the PUSH instruction (which pushes its operand on the stack) when executed with the CS register (part of which is the current privilege level). A guest OS could easily determine that it is not running at privilege level 0.

1.1.2 Address-space compression

Operating systems expect to have access to the processor's full virtual address space, known as the linear-address space in IA-32. A VMM must reserve for itself some portion of the guest's virtual-address space. The VMM could run entirely within the guest's virtual-address space, which allows it easy access to guest data, although the VMM's instructions and data structures might use a substantial amount of the guest's virtual-address space. Alternatively, the VMM could run in a separate address space, but even in that case the VMM must use a minimal amount of the guest's virtual-address space for the control structures

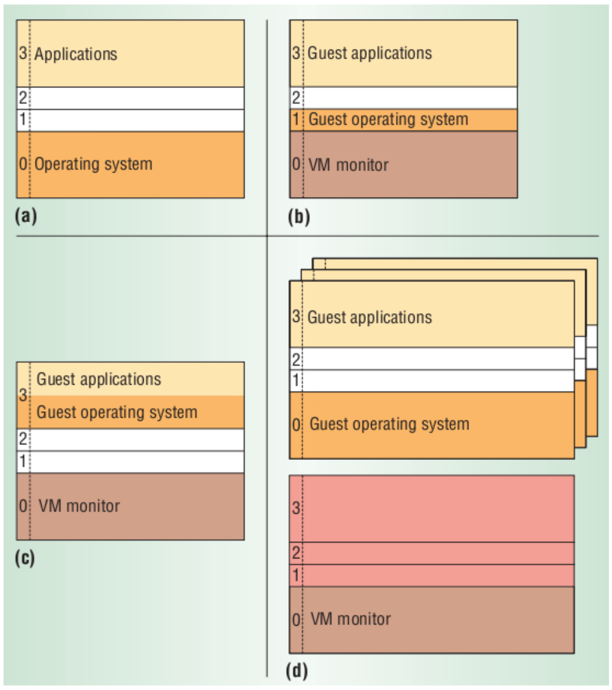


Figure 1: rings rings rings

that manage transitions between guest software and the VMM. (For IA-32, these structures include the IDT and the GDT, which reside in the linear-address space.) The VMM must prevent guest access to those portions of the guest's virtual-address space that the VMM is using. Otherwise, the VMM's integrity could be compromised if the guest can write to those portions, or the guest could detect that it is running in a virtual machine if it can read them. Guest attempts to access these portions of the address space must generate transitions to the VMM, which can emulate or otherwise support them. The term address-space compression refers to the challenges of protecting these portions of the virtual-address space and supporting guest accesses to them.

1.1.3 Nonfaulting access to privileged state

Privilege-based protection prevents unprivileged software from accessing certain components of CPU state. In most cases, attempted accesses result in faults, allowing a VMM to emulate the desired guest instruction. However, the IA-32 architecture includes instructions that access privileged state and do not fault when executed with insufficient privilege. For example, the IA-32 registers GDTR, IDTR, LDTR, and TR contain pointers to data structures that control CPU operation. Software can execute the instructions that write to, or load, these registers (LGDT, LIDT, LLDT, and LTR) only at privilege level 0. However, software can execute the instructions that read, or store, from these registers (SGDT, SIDT, SLDT, and STR) at any privilege level. If the VMM maintains these registers with unexpected values, a guest OS using the latter instructions could determine that it does not have full control of the CPU.

1.1.4 Adverse impacts on guest transitions

Ring deprivileging can interfere with the effectiveness of facilities in the IA-32 architecture that accelerate the delivery and handling of transitions to OS software. The IA-32 SYSENTER and SYSEXIT instructions support low-latency system calls. SYSENTER always effects a transition to privilege level 0, and SYSEXIT will fault if executed outside that privilege level. Ring deprivileging thus has the following implications:

- Executions of SYSENTER by a guest application will cause a transition to the VMM and not to the guest OS. The VMM must thus emulate every guest execution of SYSENTER.
- Execution of SYSEXIT by a guest OS will cause a fault to the VMM. Thus, the VMM must emulate every guest execution of SYSEXIT.

1.1.5 Interrupt virtualization

Providing support for external interrupts, especially regarding interrupt masking, presents some specific challenges to VMM design. The IA-32 architecture provides mechanisms for masking external interrupts, preventing their delivery when the OS is not ready for them. IA-32 uses the interrupt flag (IF) in the EFLAGS register to control interrupt masking. A VMM will likely manage external interrupts and deny guest software the ability to control interrupt masking. Existing protection mechanisms allow such denial of control by ensuring that guest attempts to control interrupt masking will fault in the context of ring deprivileging. Such faulting can cause problems because some operating systems frequently mask and unmask interrupts. Intercepting every guest attempt to do so could significantly affect system performance.

Even if it were possible to prevent guest modifications of interrupt masking without intercepting each attempt, challenges would remain when a VMM has a "virtual interrupt" to deliver to a guest. A virtual interrupt should be delivered only when the guest has unmasked interrupts. To deliver virtual interrupts in a timely way, a VMM should intercept some, but not all, attempts by a guest to modify interrupt masking. Doing so could signicantly complicate the design of a VMM.

1.1.6 Ring compression

Ring deprivileging uses translation privilege-based mechanisms to protect the VMM from guest software. IA-32 includes two such mechanisms: segment limits and paging. Because segment limits do not apply in 64-bit mode, paging must be used in this mode. Because IA-32 paging does not distinguish privilege levels 0-2, the guest OS must run at privilege level 3. Thus, the guest OS will run at the same privilege level as guest applications and will not be protected from them. This problem is called ring compression.

1.1.7 Access to hidden state

Some components of IA-32 CPU state are not represented in any software-accessible register. Examples include the hidden descriptor caches for the segment registers. A segment-register load copies a referenced descriptor (from the GDT or LDT) into this cache, which is not modified if software later writes to the descriptor tables. IA-32 does not provide mechanisms for saving and restoring these hidden components of a guest context when changing VMs or for preserving them while the VMM is running.

1.2 Addressing virtualization challenges in software

To address the virtualization challenges that the IA-32 architecture presents, VMM designers have developed creative solutions that modify guest software (source or binary). There are examples of VMMs that use sourcelevel modifiations in a technique called paravirtualization. Developers of these VMMs modify a guest-OS kernel and its device drivers to create an interface that is easier to virtualize. Paravirtualization offers high performance and does not require making changes to guest applications. A disadvantage of paravirtualization is that it limits the range of supported operating systems. For example, Xen cannot currently support an operating system that its developers have not modified, such as Microsoft Windows.

A VMM can support legacy operating systems by making modifications directly to guest-OS binaries. VMMs that use such binary translation techniques include those developed by VMware as well as Virtual PC and Virtual Server from Microsoft. Such VMMs support a broader range of operating systems, albeit with higher performance overheads, than VMMs that use paravirtualization.

A central design goal for Intel Virtualization Technology is to eliminate the need for CPU paravirtualization and binary translation techniques and thereby enable the implementation of VMMs that can support a broad range of unmodified guest operating systems while maintaining high levels of performance.

1.3 Intel Virtualization Technology

This section describes the basics of virtual machine architecture and an overview of the virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments.

1.3.1 Virtual Machine Architecture

Virtual-machine extensions define processor-level support for virtual machines on IA-32 processors. Two principal classes of software are supported:

- Virtual-machine monitors (VMM): A VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software (see next paragraph) with an abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.

- Guest software: Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each operates independently of other virtual machines and uses on the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software stack acts as if it were running on a platform with no VMM. Software executing in a virtual machine must operate with reduced privilege so that the VMM can retain control of platform resources.

1.3.2 Introduction to VMX operation

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits.

Processor behavior in VMX root operation is very much as it is outside VMX operation. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited.

Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources. There is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine. Because VMX operation places restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

1.3.3 Life Cycle of VMM software

Figure 2 illustrates the life cycle of a VMM and its guest software as well as the interactions between them. The following items summarize that life cycle:

- Software enters VMX operation by executing a VMXON instruction.

- Using VM entries, a VMM can then enter guests into virtual machines (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.

- VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the virtual machine using a VM entry.

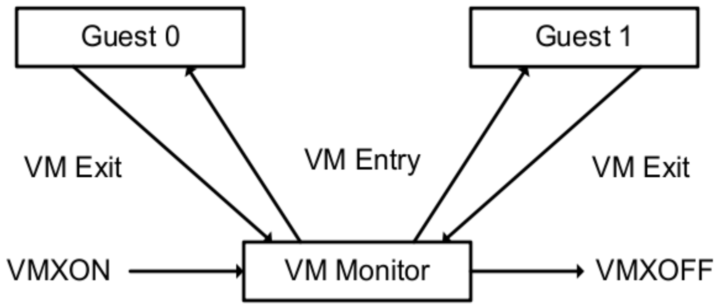


Figure 2: Interaction of a Virtual-Machine Monitor and Guests

- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

1.3.4 Virtual Machine Control Structure

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual-machine control structure (VMCS). Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions. A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

1.3.5 Restrictions on VMX operation

VMX operation places restrictions on processor operation. These are detailed below:

- In VMX operation, processors may fix certain bits in CR0 and CR4 to specific values and not support other values. VMXON fails if any of these bits contains an unsupported value. Any attempt to set one of these bits to an unsupported value while in VMX operation (including VMX root operation) using any of the CLTS, LMSW, or MOV CR instructions causes a general-protection exception. VM entry or VM exit cannot set any of these bits to an unsupported value.(2)

  NOTE The first processors to support VMX operation require that the following bits be 1 in VMX operation: CR0.PE, CR0.NE, CR0.PG, and CR4.VMXE. The restrictions on CR0.PE and CR0.PG imply that VMX operation is supported only in paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in unpaged protected mode or in real-address mode natively. But there are techniques to support these kind of guests with vt-x.

- VMXON fails if a logical processor is in A20M mode. Once the processor is in VMX operation, A20M interrupts are blocked. Thus, it is impossible to be in A20M mode in VMX operation.
- The INIT signal is blocked whenever a logical processor is in VMX root operation. It is not blocked in VMX non-root operation. Instead, INITs cause VM exits.

2 Virtual Machine Control Structure

2.1 Overview

The virtual-machine control data structure (VMCS) is defined for VMX operation. A VMCS manages transitions in and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor. Each logical processor associates a region in memory with each VMCS. This region is called the VMCS region. Software references a specific VMCS by using the 64-bit physical address of the region; such an address is called a VMCS pointer. VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). A logical processor may maintain any number of active VMCSs. At any given time, one is the current VMCS:

- Software makes a VMCS active by executing VMPTRLD with the address of the VMCS. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. Software should not make a VMCS active on more than one logical processor. Software makes a VMCS inactive by executing VMCLEAR with the address of the VMCS. A logical processor does not use an inactive VMCS or maintain its state on the processor.
- Software makes a VMCS current by executing VMPTRLD with the address of the VMCS; that address is loaded into the current-VMCS pointer. VMX instructions VMLAUNCH, VMPTRST, VMREAD, VMRESUME, and VMWRITE operate on the current VMCS. A VMCS remains current until either software executes VMPTRLD with the address of a different VMCS (which then becomes the current VMCS) or software executes VMCLEAR with the address of the current VMCS (after which there is no current VMCS).

NOTE: This document uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.).

2.2 Format of the VMCS region

A VMCS region comprises up to 4-KBytes.

The first 32 bits of the VMCS region contain the VMCS revision identifier. Processors that maintain VMCS data in different formats use different VMCS revision identifiers. These identifiers enable software to avoid using a VMCS region formatted for one processor on a processor that uses a different format. Software should write the VMCS revision identifier to the VMCS region before using that region for a VMCS. The VMCS revision identifier is never written by the processor; VMPTRLD may fail if its operand references a VMCS region whose VMCS revision identifier differs from that used by the processor. Software can discover the VMCS revision identifier that a processor uses by reading the VMX capability MSR IA32_VMX_BASIC.

The next 32 bits of the VMCS region are used for the VMX-abort indicator. The contents of these bits do not control processor operation in any way. A logical processor writes a non-zero value into these bits if a VMX abort occurs. Software may also write into this field.

The remainder of the VMCS region is used for VMCS data (those parts of the VMCS that control VMX non-root operation and the VMX transitions). The format of these data is implementation-specific. To ensure proper behavior in VMX operation, software should maintain the VMCS region and related structures in writeback cacheable memory. Future implementations may allow or require a different memory type. Software should consult the VMX capability MSR IA32_VMX_BASIC.

2.3 Organization of VMCS data

The VMCS data are organized into six logical groups:

- Guest-state area. Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
- Host-state area. Processor state is loaded from the host-state area on VM exits.
- VM-execution control fields. These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits.
- VM-exit control fields. These fields control VM exits.
- VM-entry control fields. These fields control VM entries.
- VM-exit information fields. These fields receive information on VM exits and describe the cause and the nature of VM exits. They are read-only.

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are sometimes referred to collectively as VMX controls.

2.4 Guest-State Area

This section describes fields contained in the guest-state area of the VMCS. As noted earlier, processor state is loaded from these fields on every VM entry and stored into these fields on every VM exit.

2.4.1 Guest Register State

The following fields in the guest-state area correspond to processor registers:

- Control registers CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- Debug register DR7 (64 bits; 32 bits on processors that do not support Intel 64 architecture).
- RSP, RIP, and RFLAGS (64 bits each; 32 bits on processors that do not support Intel 64 architecture).5
- The following fields for each of the registers CS, SS, DS, ES, FS, GS, LDTR, and TR:
  - Selector (16 bits).
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture). The base-address fields for CS, SS, DS, and ES have only 32 architecturally-defined bits; nevertheless, the corresponding VMCS fields have 64 bits on processors that support Intel 64 architecture.
  - Segment limit (32 bits). The limit field is always a measure in bytes.
  - Access rights (32 bits). The format of this field is given in Table 20-2 and detailed as follows:

  The base address, segment limit, and access rights compose the "hidden" part (or "descriptor cache") of each segment register. These data are included in the VMCS because it is possible for a segment register's descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register's selector. Note that the value of the DPL field for SS is always equal to the logical processor's current privilege level (CPL).
- The following fields for each of the registers GDTR and IDTR:
  - Base address (64 bits; 32 bits on processors that do not support Intel 64 architecture).
  - Limit (32 bits). The limit fields contain 32 bits even though these fields are specified as only 16 bits in the architecture.
- The following MSRs:
  - IA32_DEBUGCTL (64 bits)
  - IA32_SYSENTER_CS (32 bits)
  - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture)
- The register SMBASE (32 bits). This register contains the base address of the logical processor's SMRAM image.

2.4.2 Guest Non-Register State

In addition to the register state just described, the guest-state area includes the following fields that characterize guest state but which do not correspond to processor registers:

- Activity state (32 bits). This field identifies the logical processor's activity state. When a logical processor is executing instructions normally, it is in the active state. Execution of certain instructions and the occurrence of certain events may cause a logical processor to transition to an inactive state in which it ceases to execute instructions. The following activity states are defined: (8)

  1. Active. The logical processor is executing instructions normally.
  2. HLT. The logical processor is inactive because it executed the HLT instruction.
  3. Shutdown. The logical processor is inactive because it incurred a triple fault (9) or some other serious error.
  4. Wait-for-SIPI. The logical processor is inactive because it is waiting for a startup-IPI (SIPI).

- Interruptibility state (32 bits). The IA-32 architecture includes features that permit certain events to be blocked for a period of time. For example, execution of STI with RFLAGS.IF = 0 blocks interrupts (and, optionally, other events) for one instruction after its execution. Another example is that execution of a MOV to SS or a POP to SS blocks interrupts for one instruction after its execution. This field contains information about such blocking.

- Pending debug exceptions (64 bits; 32 bits on processors that do not support Intel 64 architecture). IA-32 processors may recognize one or more debug exceptions without immediately delivering them. This field contains information about such exceptions.

- VMCS link pointer (64 bits). This field is included for future expansion. Software should set this field to FFFFFFFF_FFFFFFFFH to avoid VM-entry failures.

2.5 Host-State Area

This section describes fields contained in the host-state area of the VMCS. As noted earlier, processor state is loaded from these fields on every VM exit. All fields in the host-state area correspond to processor registers:

- CR0, CR3, and CR4 (64 bits each; 32 bits on processors that do not support Intel 64 architecture).

- RSP and RIP (64 bits each; 32 bits on processors that do not support Intel 64 architecture).

- Selector fields (16 bits each) for the segment registers CS, SS, DS, ES, FS, GS, and TR. There is no field in the host-state area for the LDTR selector.

- Base-address fields for FS, GS, TR, GDTR, and IDTR (64 bits each; 32 bits on processors that do not support Intel 64 architecture).
- The following MSRs:
  - IA32_SYSENTER_CS (32 bits)
  - IA32_SYSENTER_ESP and IA32_SYSENTER_EIP (64 bits; 32 bits on processors that do not support Intel 64 architecture).

In addition to the state identified here, some processor state components are loaded with fixed values on every VM exit; there are no fields corresponding to these components in the host-state area.

2.6 VM-Execution Control Fields

The VM-execution control fields govern VMX non-root operation.

2.6.1 Pin-Based VM-Execution Controls

The pin-based VM-execution controls constitute a 32-bit vector that governs the handling of asynchronous events like interrupts (some asynchronous events cause VM exits regardless of the settings of the pin-based VM-execution controls). For example, if the field named "External-interrupt exiting" is 1, external interrupts cause VM exits. Otherwise, they are delivered normally through the guest interrupt-descriptor table (IDT). If this control is 1, the value of RFLAGS.IF does not affect interrupt blocking.

The other two controls are related with NMIs and Virtual NMIs

2.6.2 Processor-Based VM-Execution Controls

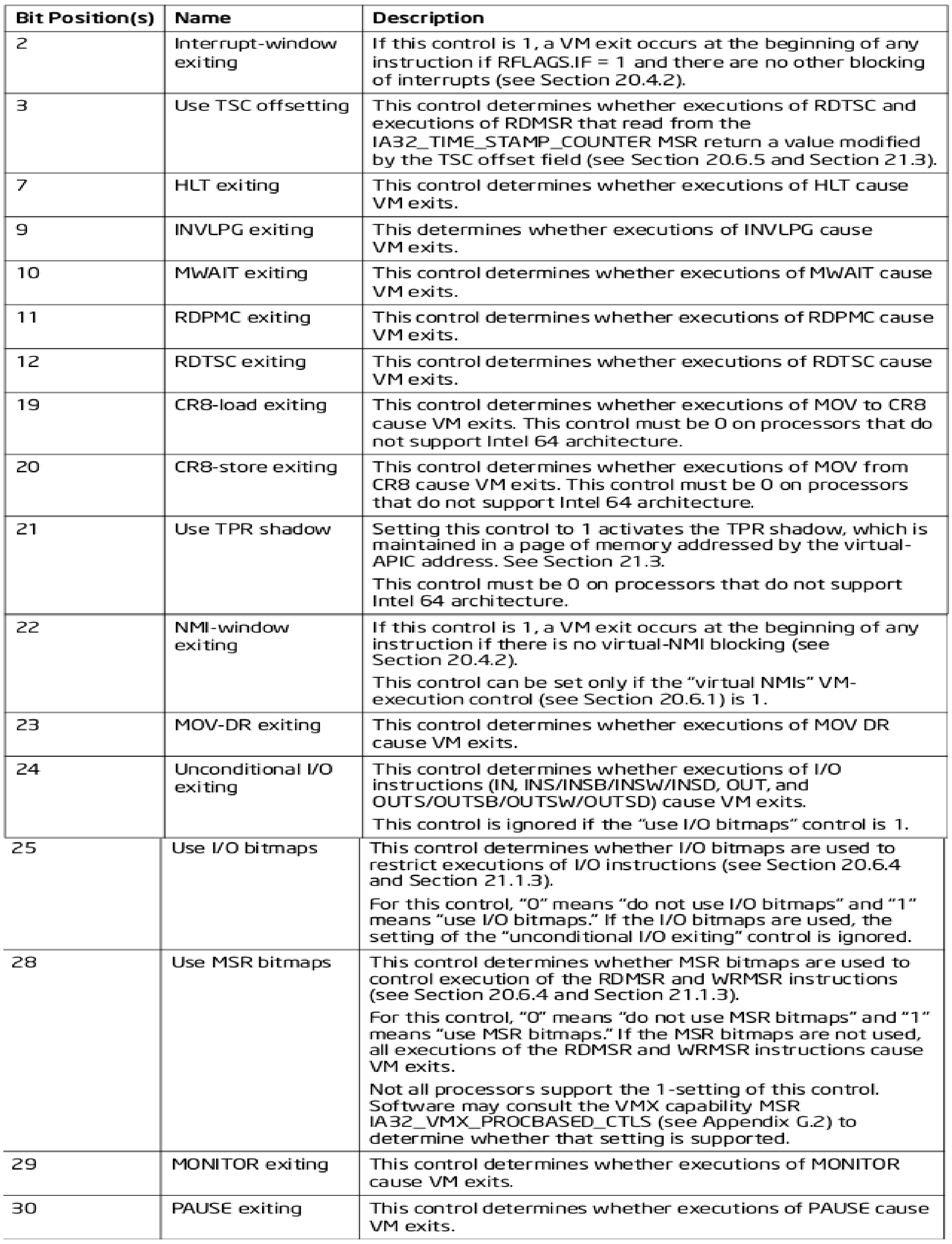
The processor-based VM-execution controls constitute a 32-bit vector that governs the handling of synchronous events, mainly those caused by the execution of specific instructions[1].

This control fields allow a VMM the flexibility to specify the instructions that cause VM exits. There are separate controls for each of the following instructions: HLT, INVLPG, MOV CR8, MOV DR, MWAIT, RDPMC, and RDTSC. These controls support a variety of virtualization strategies. It also includes the "use I/O bitmaps" and "use MSR bitmaps" fields, which indicates the use of these control bitmaps, and a "use TRP shadow" field which activates a shadow TRP maintained in a page of memory addressed by the virtual APIC address. See figure 3 for more details.

2.6.3 Exception Bitmap

The exception bitmap is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exception's vector.

[1]Some instructions cause VM exits regardless of the settings of the processor-based VM-execution controls, as do task switches.



Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 20.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 20.6.5 and Section 21.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPMC exiting	This control determines whether executions of RDPMC cause VM exits.
12	RDTSC exiting	This control determines whether executions of RDTSC cause VM exits.
19	CR8-load exiting	This control determines whether executions of MOV to CR8 cause VM exits. This control must be 0 on processors that do not support Intel 64 architecture.
20	CR8-store exiting	This control determines whether executions of MOV from CR8 cause VM exits. This control must be 0 on processors that do not support Intel 64 architecture.
21	Use TPR shadow	Setting this control to 1 activates the TPR shadow, which is maintained in a page of memory addressed by the virtual-APIC address. See Section 21.3. This control must be 0 on processors that do not support Intel 64 architecture.
22	NMI-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if there is no virtual-NMI blocking (see Section 20.4.2). This control can be set only if the "virtual NMIs" VM-execution control (see Section 20.6.1) is 1.
23	MOV-DR exiting	This control determines whether executions of MOV DR cause VM exits.
24	Unconditional I/O exiting	This control determines whether executions of I/O instructions (IN, INS/INSB/INSW/INSD, OUT, and OUTS/OUTSB/OUTSW/OUTSD) cause VM exits. This control is ignored if the "use I/O bitmaps" control is 1.
25	Use I/O bitmaps	This control determines whether I/O bitmaps are used to restrict executions of I/O instructions (see Section 20.6.4 and Section 21.1.3). For this control, "0" means "do not use I/O bitmaps" and "1" means "use I/O bitmaps." If the I/O bitmaps are used, the setting of the "unconditional I/O exiting" control is ignored.
28	Use MSR bitmaps	This control determines whether MSR bitmaps are used to control execution of the RDMSR and WRMSR instructions (see Section 20.6.4 and Section 21.1.3). For this control, "0" means "do not use MSR bitmaps" and "1" means "use MSR bitmaps." If the MSR bitmaps are not used, all executions of the RDMSR and WRMSR instructions cause VM exits. Not all processors support the 1-setting of this control. Software may consult the VMX capability MSR IA32_VMX_PROCBASED_CTLS (see Appendix G.2) to determine whether that setting is supported.
29	MONITOR exiting	This control determines whether executions of MONITOR cause VM exits.
30	PAUSE exiting	This control determines whether executions of PAUSE cause VM exits.

Figure 3: Definitions of Processor-Based VM-Execution Controls

Whether a page fault (exception with vector 14) causes a VM exit is determined by bit 14 in the exception bitmap as well as the error code produced by the page fault and two 32-bit fields in the VMCS (the page-fault error-code mask and pagefault error-code match). See section 3 for details.

2.6.4 I/O-Bitmap Addresses

The VM-execution control fields include the 64-bit physical addresses of I/O bitmaps A and B (each of which are 4 KBytes in size). I/O bitmap A contains one bit for each I/O port in the range 0000H through 7FFFH; I/O bitmap B contains bits for ports in the range 8000H through FFFFH. A logical processor uses these bitmaps if and only if the "use I/O bitmaps" control is 1. If the bitmaps are used, execution of an I/O instruction causes a VM exit if any bit in the I/O bitmaps corresponding to a port it accesses is 1.

2.6.5 Time-Stamp Counter Offset

VM-execution control fields include a 64-bit TSC-offset field. If the "RDTSC exiting" control is 0 and the "use TSC offsetting" control is 1, this field controls executions of the RDTSC instruction and executions of the RDMSR instruction that read from the IA32_TIME_STAMP_COUNTER MSR. The signed value of the TSC offset is combined with the contents of the time-stamp counter (using signed addition) and the sum is reported to guest software in EDX:EAX.

2.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4

VM-execution control fields include guest/host masks and read shadows for the CR0 and CR4 registers. These fields control executions of instructions that access those registers (including CLTS, LMSW, MOV CR, and SMSW). They are 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not. In general, bits set to 1 in a guest/host mask correspond to bits "owned" by the host:

- Guest attempts to set them (using CLTS, LMSW, or MOV to CR) to values differing from the corresponding bits in the corresponding read shadow cause VM exits. Guest reads (using MOV from CR or SMSW) return values for these bits from the corresponding read shadow.
- Bits cleared to 0 correspond to bits "owned" by the guest; guest attempts to modify them succeed and guest reads return values for these bits from the control register itself.

2.6.7 CR3-Target Controls

The VM-execution control fields include a set of 4 CR3-target values and a CR3 target count. The CR3-target values each have 64 bits on processors that support Intel 64 architecture and 32 bits on processors that do not. The CR3-target count has 32 bits on all processors. An execution of MOV to CR3 in VMX non-root operation does not cause a VM exit if its source operand matches one of these values. If the CR3-target count is n, only the first n CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit.

There are no limitations on the values that can be written for the CR3-target values. VM entry fails if the CR3-target count is greater than 4. Future processors may support a different number of CR3-target values.

2.6.8 Controls for CR8 Accesses

On processors that support Intel 64 architecture, the CR8 register can be used in 64-bit mode to access the task-priority register (TPR) of the logical processor's local APIC. The VMCS contains two fields that control MOV CR8 instructions if the "use TPR shadow" VM-execution control is 1:

- Virtual-APIC page address (64 bits). This field is the physical address of the 4-KByte virtual-APIC page. The virtual-APIC page contains the TPR shadow, which is read and written by the MOV CR8 instructions. The TPR shadow comprises bits 7:4 in byte 128 of the virtual-APIC page. If the "use TPR shadow" VM-execution control is 1, the virtual-APIC page address must be 4-KByte aligned.
- TPR threshold (32 bits). Bits 3:0 of this field determine the threshold below which the TPR shadow (see previous item) cannot fall. A VM exit occurs after an execution of MOV to CR8 that reduces the TPR shadow below this value.

  These fields exist only on processors that support the 1-setting of the "use TPR shadow" VM-execution control. Note that the TPR in the local APIC can also be accessed using memory-mapped I/O. These controls does not affect accesses made in that way. They affect only MOV CR8 instructions.

2.6.9 MSR-Bitmap Address

On processors that support the 1-setting of the "use MSR bitmaps" VM-execution control, the VM-execution control fields include the 64-bit physical address of four contiguous MSR bitmaps, which are each 1-KByte in size. This field does not exist on processors that do not support the 1-setting of that control. The four bitmaps are:

- Read bitmap for low MSRs (located at the MSR-bitmap address). This contains one bit for each MSR address in the range 00000000H – 00001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- Read bitmap for high MSRs (located at the MSR-bitmap address plus 1024). This contains one bit for each MSR address in the range C0000000H – C0001FFFH. The bit determines whether an execution of RDMSR applied to that MSR causes a VM exit.
- Write bitmap for low MSRs (located at the MSR-bitmap address plus 2048). This contains one bit for each MSR address in the range 00000000H – 00001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

- Write bitmap for high MSRs (located at the MSR-bitmap address plus 3072). This contains one bit for each MSR address in the range C0000000H – C0001FFFH. The bit determines whether an execution of WRMSR applied to that MSR causes a VM exit.

2.6.10 Executive-VMCS Pointer

The executive-VMCS pointer is a 64-bit field used in the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). SMM VM exits save this field. VM entries that return from SMM use this field.

2.7 VM-Exit Control Fields

The VM-exit control fields govern the behavior of VM exits.

2.7.1 VM-Exit Controls

The VM-exit controls constitute a 32-bit vector that governs the basic operation of VM exits. There are a field related with the host address-space size (whether the host should be woking in 64-bit mode after a VM exit) and a field that indicates if the logical processor acknowledges the interrupt controller, acquiring the interrupt's vector, during a VM exit due to external interrupts.

2.7.2 VM-Exit Controls for MSRs

A VMM may specify lists of MSRs to be stored and loaded on VM exits.

The following VM-exit control fields determine how MSRs are stored on VM exits:

- VM-exit MSR-store count (32 bits). This field specifies the number of MSRs to be stored on VM exit.

- VM-exit MSR-store address (64 bits). This field contains the physical address of the VM-exit MSR-store area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-store count.

The following VM-exit control fields determine how MSRs are loaded on VM exits:

- VM-exit MSR-load count (32 bits). This field contains the number of MSRs to be loaded on VM exit. It is recommended that this count not exceed 512 bytes. Otherwise, unpredictable processor behavior (including a machine check) may result during VM exit.

- VM-exit MSR-load address (64 bits). This field contains the physical address of the VM-exit MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-exit MSR-load count. If the VM-exit MSR-load count is not zero, the address must be 16-byte aligned.

2.8 VM-Entry Control Fields

The VM-entry control fields govern the behavior of VM entries.

2.8.1 VM-Entry Controls

The VM-entry controls constitute a 32-bit vector that governs the basic operation of VM entries.

There is a control that determines whether the logical processor is in IA-32e mode after VM entry, on processors that support Intel 64 architecture. Another control determines whether the logical processor is in system-management mode (SMM) after VM entry.

2.8.2 VM-Entry Controls for MSRs

A VMM may specify a list of MSRs to be loaded on VM entries. The following VM-entry control fields manage this functionality:

- VM-entry MSR-load count (32 bits). This field contains the number of MSRs to be loaded on VM entry.
- VM-entry MSR-load address (64 bits). This field contains the physical address of the VM-entry MSR-load area. The area is a table of entries, 16 bytes per entry, where the number of entries is given by the VM-entry MSR-load count.

2.8.3 VM-Entry Controls for Event Injection

VM entry can be configured to conclude by delivering an event through the guest IDT (after all guest state and MSRs have been loaded). This process is called event injection and is controlled by the following three VM-entry control fields:

- VM-entry interruption-information field (32 bits). This field provides details about the event to be injected:
  - The vector (bits 7:0) determines which entry in the IDT is used.
  - The interruption type (bits 10:8) determines details of how the injection is performed. It could be an external interrupt, a NMI, a hardware exception, a software interrupt, etc.

    In general, a VMM should use the type hardware exception for all exceptions other than breakpoint exceptions and overflow exceptions; it should use the type software exception for those.
  - For exceptions, the deliver-error-code bit (bit 11) determines whether delivery pushes an error code on the guest stack.
  - VM entry injects an event if and only if the valid bit (bit 31) is 1.
- VM-entry exception error code (32 bits). This field is used if and only if the valid bit (bit 31) and the deliver-error-code bit (bit 11) are both set in the VM-entry interruption-information field.

- VM-entry instruction length (32 bits). For injection of events whose type is software interrupt, software exception, or privileged software exception, this field is used to determine the value of RIP that is pushed on the stack.

2.9 VM-Exit Information Fields

The VMCS contains a section of read-only fields that contain information about the most recent VM exit.

2.9.1 Basic VM-Exit Information

The following VM-exit information fields provide basic information about a VM exit:

- Exit reason (32 bits). This field encodes the reason for the VM exit.
  - Bits 15:0 provide basic information about the cause of the VM exit (if bit 31 is clear) or of the VM-entry failure (if bit 31 is set).
  - Bit 29 is set if and only if the processor was in VMX root operation at the time the VM exit occurred. This can happen only for SMM VM exits.
  - Because some VM-entry failures load processor state from the host-state area, software must be able to distinguish such cases from true VM exits. Bit 31 is used for that purpose.
- Exit qualification (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field contains additional information about the cause of VM exits due to the following: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); task switches; INVLPG; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; control-register accesses; MOV DR; I/O instructions; and MWAIT. The format of the field depends on the cause of the VM exit.

2.9.2 Information for VM Exits Due to Vectored Events

Event-specific information is provided for VM exits due to the following vectored events: exceptions (including those generated by the instructions INT3, INTO, BOUND, and UD2); external interrupts that occur while the “acknowledge interrupt on exit” VM-exit control is 1; and non-maskable interrupts (NMIs). This information is provided in the following fields:

- VM-exit interruption information (32 bits). This field receives basic information associated with the event causing the VM exit, like the vector of interruption or exception, the type of the interruption (external interrupt, NMI, software exception, etc), validity of error code, etc.
- VM-exit interruption error code (32 bits). For VM exits caused by hardware exceptions that would have delivered an error code on the stack, this field receives that error code.

2.9.3 Information for VM Exits Due to Instruction Execution

The following fields are used for VM exits caused by attempts to execute certain instructions in VMX non-root operation:

- VM-exit instruction length (32 bits). For VM exits resulting from instruction execution, this field receives the length in bytes of the instruction whose execution led to the VM exit.
- Guest linear address (64 bits; 32 bits on processors that do not support Intel 64 architecture). This field is used in the following cases:
  - VM exits due to attempts to execute LMSW with a memory operand.
  - VM exits due to attempts to execute INS or OUTS.
  - VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions.
- VMX-instruction information (32 bits). For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, or VMXON, this field receives details about the instruction that caused the VM exit.

3 VMX non-root operation

In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment.

This section describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation).

3.1 Instructions that cause VM exits

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are "fault-like," meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction.

3.1.1 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, INVD, MOV from CR3. This is also true of instructions introduced with VMX, which include: VMCALL,2 VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.

3.1.2 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:

- CLTS. The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- HLT. The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD. The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:
  - If both controls are 0, the instruction executes normally.
  - If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.
  - If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap. If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).
- INLVPG. The INLVPG instruction causes a VM exit if the "INLVPG exiting" VM-execution control is 1.
- LMSW. In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. Note that LMSW never clears bit 0 of CR0 (CR0.PE). Thus, LMSW causes a VM exit if either of the following are true:
  - The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
- MONITOR. The MONITOR instruction causes a VM exit if the "MONITOR exiting" VM-execution control is 1.
- MOV from CR8. The MOV from CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) causes a VM exit if the "CR8-store exiting" VM-execution control is 1.

- MOV to CR0. The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
- MOV to CR3. The MOV to CR3 instruction causes a VM exit unless the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Note that, if the CR3-target count is n, only the first n CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit.
- MOV to CR4. The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- MOV to CR8. The MOV to CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) causes a VM exit if the “CR8-load exiting” VM-execution control is 1. Note that, if this control is 0, the behavior of the MOV to CR8 instruction is modified if the “use TPR shadow” VM-execution control is 1 and it may cause a trap-like VM exit.
- MOV DR. The MOV DR instruction causes a VM exit if the “MOV-DR exiting” VM-execution control is 1.
- MWAIT. The MWAIT instruction causes a VM exit if the “MWAIT exiting” VM-execution control is 1.
- PAUSE. The PAUSE instruction causes a VM exit if the “PAUSE exiting” VM-execution control is 1.
- RDMSR. The RDMSR instruction causes a VM exit if any of the following are true:
  - The “use MSR bitmaps” VM-execution control is 0.
  - The value of RCX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
  - The value of RCX is in the range 00000000H – 00001FFFH and the nth bit in read bitmap for low MSRs is 1, where n is the value of RCX.
  - The value of RCX is in the range C0000000H – C0001FFFH and the nth bit in read bitmap for high MSRs is 1, where n is the value of RCX & 00001FFFH.
- RDPMC. The RDPMC instruction causes a VM exit if the “RDPMC exiting” VM-execution control is 1.
- RDTSC. The RDTSC instruction causes a VM exit if the “RDTSC exiting” VM-execution control is 1.
- RSM. The RSM instruction causes a VM exit if executed in system-management mode (SMM).

- WRMSR. The WRMSR instruction causes a VM exit if any of the following are true:
  - The "use MSR bitmaps" VM-execution control is 0.
  - The value of RCX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
  - The value of RCX is in the range 00000000H – 00001FFFH and the nth bit in write bitmap for low MSRs is 1, where n is the value of RCX.
  - The value of RCX is in the range C0000000H – C0001FFFH and the nth bit in write bitmap for high MSRs is 1, where n is the value of RCX & 00001FFFH.
- The MOV to CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) may cause a "trap-like" VM exit. This means that the instruction completes before the VM exit occurs and that processor state is updated by the instruction (for example, the value of RIP saved in the guest-state area of the VMCS references the next instruction). Specifically, a VM exit occurs after execution of MOV to CR8 if the following are true:
  - The "CR8-load exiting" VM-execution control is 0.
  - The "use TPR shadow" VM-execution control is 1.
  - The execution of MOV to CR8 reduces the value of the TPR shadow below that of the TPR threshold.

3.2 Other causes of VM exits

In addition to VM exits caused by instruction execution, the following events can cause VM exits:

- Exceptions. Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap. If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT3, INTO, BOUND, and UD2.

  Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a logical processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if PFEC & PFEC_MASK = PFEC_MATCH. If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear). Thus, if the design requires VM exits on all page faults, software can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If the design does not require VM exits on page faults, software could set bit

14 in the exception bitmap to 1, set the page-fault error-code mask field to 00000000H, and set the page-fault error-code match field to FFFFFFFFH.

- External interrupts. An external interrupt causes a VM exit if the "external interrupt exiting" VM-execution control is 1. Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)

- Non-maskable interrupts (NMIs). An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)

- INIT signals. INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)

- Start-up IPIs (SIPIs). SIPIs cause VM exits. If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)

- Task switches. Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit.

In addition, there is one control that causes VM exits based on the readiness of guest software to receive an external interrupt:

- If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS. Such a VM exit occurs immediately after VM entry if the above conditions are true. Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

- If the "NMI-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS. (A logical processor may also prevent such a VM exit if there is blocking of events by STI.) Such a VM exit occurs immediately after VM entry if the above conditions are true. Debug-trap exceptions and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over nonmaskable interrupts (NMIs) and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.

3.3 Changes to instruction behavior in VMX non-root operation

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

- CLTS. Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation, in which case CLTS causes a general-protection exception.
  - If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.
- IRET. Behavior of IRET with regard to NMI blocking is determined by the settings of the "NMI exiting" and "virtual NMIs" VM-execution controls:
  - If the "NMI exiting" VM-execution control is 0, IRET operates normally and unblocks NMIs.
  - If the "NMI exiting" VM-execution control is 1, IRET does not affect blocking of NMIs.
  - If the "virtual NMIs" VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking. If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0.
- LMSW. An execution of LMSW that does not cause a VM exit leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. It causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation.
- MOV from CR0. The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host

mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow. Note that, depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

- MOV from CR4. The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow. Note that, depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.
- MOV from CR8. Behavior of the MOV from CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) is determined by the settings of the "CR8-store exiting" and "use TPR shadow" VM-execution controls:
  - If both controls are 0, MOV from CR8 operates normally.
  - If the "CR8-store exiting" VM-execution control is 0 and the "use TPR shadow" VM-execution control is 1, MOV from CR8 reads from the TPR shadow. Specifically, it loads bits 3:0 of its destination operand with the value of bits 7:4 of byte 128 of the page referenced by the virtual-APIC page address. Bits 63:4 of the destination operand are cleared.
  - If the "CR8-store exiting" VM-execution control is 1, MOV from CR8 causes a VM exit; the "use TPR shadow" VM-execution control is ignored in this case.
- MOV to CR0. An execution of MOV to CR0 that does not cause a VM exit leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. It causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation.
- MOV to CR4. An execution of MOV to CR4 that does not cause a VM exit leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit to a value not supported in VMX operation.
- MOV to CR8. Behavior of the MOV to CR8 instruction (which can be executed only in 64-bit mode on processors that support Intel 64 architecture) is determined by the settings of the "CR8-load exiting" and "use TPR shadow" VM-execution controls:
  - If both controls are 0, MOV to CR8 operates normally.

  - If the "CR8-load exiting" VM-execution control is 0 and the "use TPR shadow" VM-execution control is 1, MOV to CR8 writes to the TPR shadow. Specifically, it stores bits 3:0 of its source operand into bits 7:4 of byte 128 of the page referenced by the virtual-APIC page address; bits 3:0 of that byte and bytes 129-131 of that page are cleared. Such a store may cause a VM exit to occur after it completes.
  - If the "CR8-load exiting" VM-execution control is 1, MOV to CR8 causes a VM exit; the "use TPR shadow" VM-execution control is ignored in this case.
  - RDMSR. If an execution of RDMSR does not cause a VM exit and if RCX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the RDMSR instruction is determined by the setting of the "use TSC offsetting" VM-execution control as well as the TSC offset:
  - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
  - If the control is 1, RDMSR loads EAX:EDX with the sum (using signed addition) of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset (interpreted as a signed value).

- RDTSC. Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls as well as the TSC offset:
  - If both controls are 0, RDTSC operates normally.
  - If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, RDTSC loads EAX:EDX with the sum (using signed addition) of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset (interpreted as a signed value).
  - If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit.

- SMSW. The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow. Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

3.4 Other Changes in VMX non-root operation

Treatments of event blocking and of task switches differ in VMX non-root operation as described in the following sections.

3.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the "external-interrupt exiting" VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0). If the "external-interrupt exiting" VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementationspecific).
- If the "NMI exiting" VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation specific).

3.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:

   (a) If CALL, INT n, or JMP accesses a task gate in IA-32e mode, a general protection exception occurs.

   (b) If CALL, INT n, INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.

   (c) If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general protection exception occurs.

   (d) If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.

   (e) If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.

   (f) If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.

(g) If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs. h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.

2. Checks are made on the new TSS selector (for example, that is within GDT limits).
3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).
4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.

Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some logical processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other logical processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

4 Memory Virtualization

VMMs must control physical memory to ensure VM isolation and to remap guest physical addresses in host physical address space for virtualization. Memory virtualization allows the VMM to enforce control of physical memory and yet support guest OSs' expectation to manage memory address translation.

4.1 Processor Operating Modes & Memory Virtualization

Memory virtualization is required to support guest execution in various processor operating modes. This includes: protected mode with paging, protected mode with no paging, real-mode and any other transient execution modes. VMX allows guest operation in protected-mode with paging enabled and in virtual-8086 mode (with paging enabled) to support guest real-mode execution. Guest execution in transient operating modes (such as in real mode with one or more segment limits greater than 64-KByte) must be emulated by the VMM. Since VMX operation requires processor execution in protected mode with paging (through CR0 and CR4 fixed bits), the VMM may utilize paging structures to support memory virtualization. To support guest real-mode execution, the VMM may establish a simple flat page table for guest linear to host physical address mapping. Memory virtualization algorithms may also need to capture other guest operating conditions (such as guest performing A20M# address masking) to map the resulting 20-bit effective guest physical addresses.

4.2 Guest & Host Physical Address Spaces

Memory virtualization provides guest software with contiguous guest physical address space starting zero and extending to the maximum address supported by the guest virtual processor's physical address width. The VMM utilizes guest physical to host physical address mapping to locate all or portions of the guest physical address space in host memory. The VMM is responsible for the policies and algorithms for this mapping which may take into account the host system physical memory map and the virtualized physical memory map exposed to a guest by the VMM. The memory virtualization algorithm needs to accommodate various guest memory uses (such as: accessing DRAM, accessing memory-mapped registers of virtual devices or core logic functions and so forth). For example:

- To support guest DRAM access, the VMM needs to map DRAM-backed guest physical addresses to host-DRAM regions. The VMM also requires the guest to host memory mapping to be at page granularity.
- Virtual devices (I/O devices or platform core logic) emulated by the VMM may claim specific regions in the guest physical address space to locate memory mapped registers. Guest access to these virtual registers may be configured to cause page-fault induced VM-exits by marking these regions as always not present. The VMM may handle these VM exits by invoking appropriate virtual device emulation code.

4.3 Virtualizing Virtual Memory by Brute Force

VMX provides the hardware features required to fully virtualize guest virtual memory accesses. VMX allows the VMM to trap guest accesses to the PAT (Page Attribute Table) MSR and the MTRR (Memory Type Range Registers). This control allows the VMM to virtualize the specific memory type of a guest memory. The VMM may control caching by controlling the guest CR0.CRD and CR0.NW bits, as well as by trapping guest execution of the INVD instruction. The VMM can trap guest CR3 loads and stores, and it may trap guest execution of INVLPG. Because a VMM must retain control of physical memory, it must also retain control over the processor's address-translation mechanisms. Specifically, this means that only the VMM can access CR3 (which contains the base of the page directory) and can execute INVLPG (the only other instruction that directly manipulates the TLB). At the same time that the VMM controls address translation, a guest operating system will also expect to perform normal memory management functions. It will access CR3, execute INVLPG, and modify (what it believes to be) page directories and page tables.

Virtualization of address translation must tolerate and support guest attempts to control address translation. A simple-minded way to do this would be to ensure that all guest attempts to access address-translation hardware trap to the VMM where such operations can be properly emulated. It must ensure that accesses to page directories and page tables also get trapped. This may be done by protecting these in-memory structures with conventional page-based protection. The VMM can do this because it can locate the page directory because its base address is in CR3 and the VMM receives control on any change to CR3; it can locate the page tables because their base addresses are in the page directory.

Such a straightforward approach is not necessarily desirable. Protection of the inmemory translation structures may be cumbersome. The VMM may maintain these structures with different values (e.g., different page base addresses) than guest software. This means that there must be traps on guest attempt to read these structures and that the VMM must maintain, in auxiliary data structures, the values to return to these reads. There must also be traps on modifications to these structures even if the translations they effect are never used. All this implies considerable overhead that should be avoided.

4.4 Alternate Approach to Memory Virtualization

Guest software is allowed to freely modify the guest page-table hierarchy without causing traps to the VMM. Because of this, the active page-table hierarchy might not always be consistent with the guest hierarchy. Any potential problems arising from inconsistencies can be solved using techniques analogous to those used by the processor and its TLB.

This section describes an alternative approach that allows guest software to freely access page directories and page tables. Traps occur on CR3 accesses and executions of INVLPG. They also occur when necessary to ensure that guest modifications to the translation structures actually take effect. The software mechanisms to support this approach are collectively called virtual TLB. This is because they emulate the functionality of the processor's physical translation look-aside buffer (TLB). The basic idea behind the virtual TLB is similar to that behind the processor TLB. While the page-table hierarchy defines the relationship between physical to linear address, it does not directly control the address translation of each memory access. Instead, translation is controlled by the TLB, which is occasionally filled by the processor with translations derived from the page-table hierarchy. With a virtual TLB, the page-table hierarchy established by guest software (specifically, the guest operating system) does not control translation, either directly or indirectly. Instead, translation is controlled by the processor (through its TLB) and by the VMM (through a page-table hierarchy that it maintains). Specifically, the VMM maintains an alternative page-table hierarchy that effectively caches translations derived from the hierarchy maintained by guest software. The remainder of this document refers to the former as the active page-table hierarchy (because it is referenced by CR3 and may be used by the processor to load its TLB) and the latter as the guest page-table hierarchy (because it is maintained by guest software). The entries in the active hierarchy may resemble the corresponding entries in the guest hierarchy in some ways and may differ in others. Guest software is allowed to freely modify the guest page-table hierarchy without causing VM exits to the VMM. Because of this, the active page-table hierarchy might not always be consistent with the guest hierarchy. Any potential problems arising from any inconsistencies can be solved using techniques analogous to those used by the processor and its TLB. Note the following:

- Suppose the guest page-table hierarchy allows more access than active hierarchy (for example: there is a translation for a linear address in the guest hierarchy but not in the active hierarchy); this is analogous to a situation in which the TLB allows less access than the page-table hierarchy. If an access occurs that would be allowed by the guest hierarchy but not the active one, a page fault occurs; this is analogous to a TLB miss. The VMM gains control (as it handles all page faults) and can update the active page-table hierarchy appropriately; this corresponds to a TLB fill.

- Suppose the guest page-table hierarchy allows less access than the active hierarchy; this is analogous to a situation in which the TLB allows more access than the page-table hierarchy. This situation can occur only if the guest operating system has modified a page-table entry to reduce access (for example: by marking it not-present). Because the older, more permissive translation may have been cached in the TLB, the processor is architecturally permitted to use the older translation and allow more access. Thus, the VMM may (through the active page-table hierarchy) also allow greater access. For the new, less permissive translation to take effect, guest software should flush any older translations from the TLB either by executing INVLPG or by loading CR3. Because both these operations will cause a trap to the VMM, the VMM will gain control and can remove from the active page-table hierarchy the translations indicated by guest software (the translation of a specific linear address for INVLPG or all translations for a load of CR3).

As noted previously, the processor reads the page-table hierarchy to cache translations in the TLB. It also writes to the hierarchy to main the accessed (A) and dirty (D) bits in the PDEs and PTEs. The virtual TLB emulates this behavior as follows:

- When a page is accessed by guest software, the A bit in the corresponding PTE (or PDE for a 4-MByte page) in the active page-table hierarchy will be set by the processor (the same is true for PDEs when active page tables are accessed by the processor). For guest software to operate properly, the VMM should update the A bit in the guest entry at this time. It can do this reliably if it keeps the active PTE (or PDE) marked not-present until it has set the A bit in the guest entry.

- When a page is written by guest software, the D bit in the corresponding PTE (or PDE for a 4-MByte page) in the active page-table hierarchy will be set by the processor. For guest software to operate properly, the VMM should update the D bit in the guest entry at this time. It can do this reliably if it keeps the active PTE (or PDE) marked read-only until it has set the D bit in the guest entry.

5 Handling interruptions in VMM

5.1 VMX support for handling interrupts

The following bullets summarize VMX support for handling interrupts:

- Control of Processor Exceptions. The VMM can get control on specific guest exceptions through the exception-bitmap in the guest controlling-VMCS.

- Control over Triple-faults. If a fault occurs while attempting to call a doublefault handler in the guest and that fault is not configured to cause a VM exit in the exception bitmap, the resulting triple fault causes a VM exit.

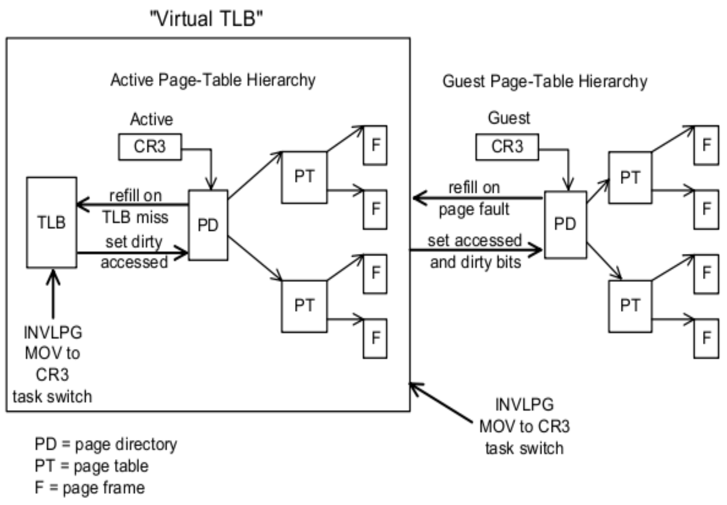


Figure 4: Virtual TLB Scheme

- Control of External-Interrupts. VMX allows both host and guest control of external interrupts through the "external-interrupt exiting" VM execution control. With guest control (external-interrupt exiting set to 0), external-interrupts do not cause VM exits and the interrupt delivery is masked by the guest programmed RFLAGS.IF value. With host control (external-interrupt exiting set to 1), external-interrupts causes VM exits and are not masked by RFLAGS.IF. The VMM can identify VM exits due to external interrupts by checking the exit-reason for an 'external-interrupt'.

- Control of Other Events. There is a pin-based VM-execution control that controls system behavior (exit or no-exit) for NMI events. Most VMM usages will need handling of NMI external events in the VMM and hence will specify host control of these events. Some processors also support a pin-based VM-execution control called "virtual NMIs." When this control is set, NMIs cause VM exits, but the processor tracks guest readiness for virtual NMIs. This control interacts with the "NMI-window exiting" VM-execution control (see below). INIT and SIPI events always cause VM exits.

- Acknowledge-Interrupt-On-Exit. The acknowledge-interrupt-on-exit bit in the VM-exit control field in the controlling-VMCS controls processor behavior for external interrupt acknowledgement. If the control bit is set, the processor acknowledges the interrupt controller to acquire the interrupt vector upon VM exit, and stores the vector in the VM-exit interruption-information field. If the control bit is clear, the external interrupt is not acknowledged during VM exit. Since RFLAGS.IF is automatically cleared on VM exits due to external interrupts, VMM re-enabling of interrupts (setting RFLAGS.IF = 1) initiates the external interrupt acknowledgement and vectoring of the external interrupt through the monitor/host

IDT.

- Event Masking Support. VMX captures the masking conditions of specific events while in VMX non-root operation through the interruptibility-state field in the guest-state area of the VMCS. This feature allows proper virtualization of various interrupt blocking states, such as: (a) blocking of external interrupts for the instruction following STI; (b) blocking of interrupts for the instruction following a MOV-SS or POP-SS instruction; (c) SMI blocking of subsequent SMIs until the next execution of RSM; and (d) NMI/SMI blocking of NMIs until the next execution of IRET or RSM. INIT and SIPI events are treated specially. INIT assertions are always blocked in VMX root operation and while in SMM, and unblocked otherwise. SIPI events are always blocked in VMX root operation. The interruptibility state is loaded from the VMCS guest-state area on every VM entry and saved into the VMCS on every VM exit.

- Event injection. VMX operation allows injecting interruptions to a guest virtual machine through the use of VM-entry interrupt-information field in VMCS. Injectable interruptions include external interrupts, NMI, processor exceptions, software generated interrupts, and software traps. If the interrupt-information field indicates a valid interrupt, exception or trap event upon the next VM entry; the processor will use the information in the field to vector a virtual interruption through the guest IDT after all guest state and MSRs are loaded. Delivery through the guest IDT emulates vectoring in non-VMX operation by doing the normal privilege checks and pushing appropriate entries to the guest stack (entries may include RFLAGS, EIP and exception error code). A VMM with host control of NMI and external interrupts can use the event-injection facility to forward virtual interruptions to various guest virtual machines.

- Interrupt-window Exiting. The interrupt-window exiting control bit in the VM-execution controls causes VM exits when guest RFLAGS.IF is 1 and no other conditions block external interrupts. If the control is 1, a VM exit occurs at the beginning of any instruction at which RFLAGS.IF = 1 and on which the interruptibility state of the guest would allow delivery of an interrupt. For example: when the guest executes an STI instruction, RFLAGS = 1, and if at the completion of next instruction the interruptibility state masking due to STI is removed; a VM exit occurs if interrupt-window exiting control is 1. The interrupt-window exiting feature allows a VMM to queue a virtual interrupt to the guest when the guest is not in an interruptible state. The VMM can set the interrupt-window exiting control for the guest and depend on a VM exit to know when the guest becomes interruptible (and, therefore, when it can inject a virtual interrupt). The VMM can detect such VM exits by checking for the basic exit reason 'interrupt-window' (value = 7). Without interrupt-window exiting support, the VMM will need to poll and check the interruptibility state of the guest to deliver virtual interrupts.

- NMI-window Exiting. If the "virtual NMIs" VM-execution is set, the processor tracks virtual-NMI blocking. The NMI-window exiting control bit in VM-execution controls causes VM exits when there is no virtual-NMI blocking. For example, after execution of the IRET instruction, a

VM exit occurs if NMIwindow exiting control is 1. The NMI-window exiting feature allows a VMM to queue a virtual NMI to a guest when the guest is not ready to receive NMIs. The VMM can set the NMI-window exiting control for the guest and depend on a VM exit to know when the guest becomes ready for NMIs (and, therefore, when it can inject a virtual NMI). The VMM can detect such VM exits by checking for the basic exit reason 'NMI window' (value = 8). Without NMI-window exiting support, the VMM will need to poll and check the interruptibility state of the guest to deliver virtual NMIs.

- VM-Exit Information. The VM-exit information fields provide details on VM exits due to exceptions and interrupts. This information is provided through the exit-qualification, VM-exit-interruption-information, instruction-length and interruption-error-code fields. Also, for VM exits that occur in the course of vectoring through the guest-IDT, information about the event that was being vectored through the guest-IDT is provided in the IDT-vectoring-information and IDT-vectoring-error-code fields. These information fields allow the VMM to identify the exception cause and to handle it properly.

5.2 External interrupt virtualization

VMX operation allows both host and guest control of external interrupts. While guest control of external interrupts might be suitable for partitioned usages (different CPU cores/threads and I/O devices partitioned to independent virtual machines), most VMMs built upon VMX are expected to utilize host control of external interrupts. The rest of this section describes a general host-controlled interrupt virtualization architecture for standard PC platforms through the use of VMX supported features.

With host control of external interrupts, the VMM (or the host OS in a hosted VMM model) manages the physical interrupt controllers in the platform and the interrupts generated through them. The VMM exposes software-emulated virtual interrupt controller devices (such as PIC and APIC) to each guest virtual machine instance.

5.2.1 Virtualization of Interrupt Vector Space

The Intel 64 and IA-32 architectures use 8-bit vectors of which 244 (20H - FFH) are available for external interrupts. Vectors are used to select the appropriate entry in the interrupt descriptor table (IDT). VMX operation allows each guest to control its own IDT. Host vectors refer to vectors delivered by the platform to the processor during the interrupt acknowledgement cycle. Guest vectors refer to vectors programmed by a guest to select an entry in its guest IDT. Depending on the I/O resource management models supported by the VMM design, the guest vector space may or may not overlap with the underlying host vector space.

- Interrupts from virtual devices: Guest vector numbers for virtual interrupts delivered to guests on behalf of emulated virtual devices have no direct relation to the host vector numbers of interrupts from physical devices on which they are emulated. A guest-vector assigned for a virtual

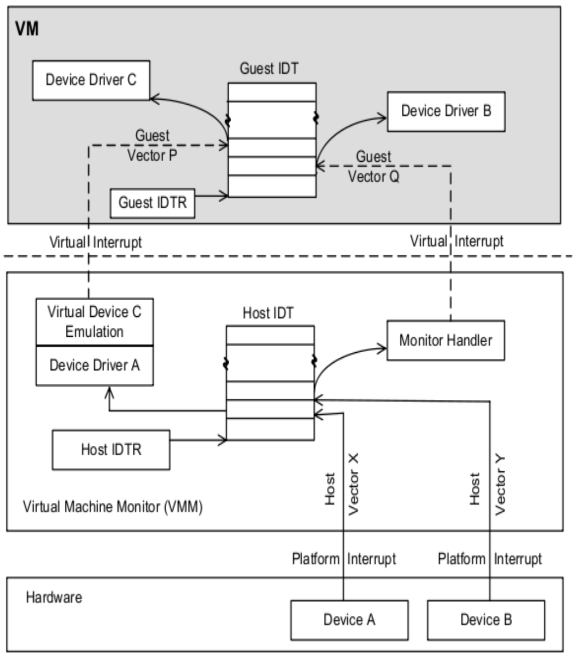


Figure 5: Host External Interrupts and Guest Virtual Interrupts

device by the guest operating environment is saved by the VMM and utilized when injecting virtual interrupts on behalf of the virtual device.

- Interrupts from assigned physical devices: Hardware support for I/O device assignment allows physical I/O devices in the host platform to be assigned (direct-mapped) to VMs. Guest vectors for interrupts from direct-mapped physical devices take up equivalent space from the host vector space, and require the VMM to perform host-vector to guest-vector mapping for interrupts.

Figure 5 illustrates the functional relationship between host external interrupts and guest virtual external interrupts. Device A is owned by the host and generates external interrupts with host vector X. The host IDT is set up such that the interrupt service routine (ISR) for device driver A is hooked to host vector X as normal. VMM emulates (over device A) virtual device C in software which generates virtual interrupts to the VM with guest expected vector P. Device B is assigned to a VM and generates external interrupts with host vector Y. The host IDT is programmed to hook the VMM interrupt service routine (ISR) for assigned devices for vector Y, and the VMM handler injects virtual interrupt with guest vector Q to the VM. The guest operating system programs the guest to hook appropriate guest driver's ISR to vectors P and Q.

5.2.2 Control of Platform Interrupts

To meet the interrupt virtualization requirements, the VMM needs to take ownership of the physical interrupts and the various interrupt controllers in the platform. VMM control of physical interrupts may be enabled through the host-control settings of the “external-interrupt exiting” VM-execution control. To take ownership of the platform interrupt controllers, the VMM needs to expose the virtual interrupt controller devices to the virtual machines and restrict guest access to the platform interrupt controllers. Intel 64 and IA-32 platforms can support three types of external interrupt control mechanisms: Programmable Interrupt Controllers (PIC), Advanced Programmable Interrupt Controllers (APIC), and Message Signaled Interrupts (MSI). The following sections provide information on the virtualization of each of these mechanisms.

PIC Virtualization Typical PIC-enabled platform implementations support dual 8259 interrupt controllers cascaded as master and slave controllers. They supporting up to 15 possible interrupt inputs. The 8259 controllers are programmed through initialization command words (ICWx) and operation command words (OCWx) accessed through specific I/O ports. The various interrupt line states are captured in the PIC through interrupt requests, interrupt service routines and interrupt mask registers. Guest access to the PIC I/O ports can be restricted by activating I/O bitmaps in the guest controlling-VMCS (activate-I/O-bitmap bit in VM-execution control field set to 1) and pointing the I/O-bitmap physical addresses to valid bitmap regions. Bits corresponding to the PIC I/O ports can be cleared to cause a VM exit on guest access to these ports. If the VMM is not supporting direct access to any I/O ports from a guest, it can set the unconditional-I/O-exiting in the VM-execution control field instead of activating I/O bitmaps. The exit-reason field in VM-exit information allows identification of VM exits due to I/O access and can provide an exit-qualification to identify details about the guest I/O operation that caused the VM exit. The VMM PIC virtualization needs to emulate the platform PIC functionality including interrupt priority, mask, request and service states, and specific guest programmed modes of PIC operation.

xAPIC Virtualization Most modern Intel 64 and IA-32 platforms include support for an APIC. While the standard PIC is intended for use on uniprocessor systems, APIC can be used in either uniprocessor or multi-processor systems. APIC based interrupt control consists of two physical components: the interrupt acceptance unit (Local APIC) which is integrated with the processor, and the interrupt delivery unit (I/O APIC) which is part of the I/O subsystem. APIC virtualization involves protecting the platform's local and I/O APICs and emulating them for the guest.

Local APIC Virtualization The local APIC is responsible for the local interrupt sources, interrupt acceptance, dispensing interrupts to the logical processor, and generating inter-processor interrupts. Software interacts with the local APIC by reading and writing its memory-mapped registers residing within a 4-KByte uncached memory region with base address stored in the IA32_APIC_BASE MSR. Since the local APIC registers are memory-mapped, the VMM can utilize memory virtualization techniques (such as page-table virtualization) to trap guest accesses to the page frame hosting the virtual local APIC registers. Local APIC virtualization in the VMM needs to emulate the various local APIC operations and registers, such as: APIC identification/format registers, the local vector table (LVT), the interrupt command register (ICR), interrupt capture registers (TMR, IRR and ISR), task and processor priority registers (TPR, PPR), the EOI register and the APIC-timer register. Since local APICs are designed to operate with non-specific EOI, local APIC emulation also needs to emulate broadcast of EOI to the guest's virtual I/O APICs for level triggered virtual interrupts. A local APIC allows interrupt masking at two levels: (1) mask bit in the local vector table entry for local interrupts and (2) raising processor priority through the TPR registers for masking lower priority external interrupts. The VMM needs to comprehend these virtual local APIC mask settings as programmed by the guest in addition to the guest virtual processor interruptibility state (when injecting APIC routed external virtual interrupts to a guest VM). VMX provides several features which help the VMM to virtualize the local APIC. These features allow many of guest TPR accesses (using CR8 only) to occur without VM exits to the VMM:

- The VMCS contains a 'Virtual-APIC page address' field. This 64-bit field is the physical address of the 4-KByte virtual APIC page (4-KByte aligned). The virtual APIC page contains a TPR shadow, which is accessed by the MOV CR8 instruction. The TPR shadow comprises bits 7:4 in byte 128 of the virtual-APIC page.
- The TPR threshold: bits 3:0 of this 32-bit field determine the threshold below which the TPR shadow cannot fall. A VM exit will occur after an execution of MOV CR8 that reduces the TPR shadow below this value.
- The processor-based VM-execution controls field contains a 'Use TPR shadow' bit and a 'CR8-store exiting' bit. If 'Use TPR shadow' is set and 'CR8-store exiting' is cleared, then a MOV from CR8 reads from the TPR shadow. If the 'CR8-store exiting' VM-execution control is set, then MOV from CR8 causes a VM exit. 'Use TPR shadow' is ignored in this case.
- The processor-based VM-execution controls field contains a 'CR8-load exiting' bit. If 'Use TPR shadow' is set and 'CR8-load exiting' is clear, then MOV to CR8 writes to the 'TPR shadow'. A VM exit will occur after this write if the value written is below the TPR threshold. If 'CR8-load exiting' is set, then MOV to CR8 causes a VM exit. 'Use TPR shadow' is ignored in this case.

I/O APIC Virtualization The I/O APIC registers are typically mapped to a 1 MByte region where each I/O APIC is allocated a 4K address window within this range. The VMM may utilize physical memory virtualization to trap guest accesses to the virtual I/O APIC memory-mapped registers. The I/O APIC virtualization needs to emulate the various I/O APIC operations and registers such as identification/version registers, indirect-I/O-access registers, EOI register, and the I/O redirection table. I/O APIC virtualization also need to emulate various redirection table entry settings such as delivery mode, destination mode, delivery status, polarity, masking, and trigger mode programmed by the guest and track remote-IRR state on guest EOI writes to various virtual local APICs.

Virtualization of Message Signaled Interrupts The PCI Local Bus Specification (Rev. 2.2) introduces the concept of message signaled interrupts (MSI). MSI enable PCI devices to request service by writing a system-specified message to a system specified address. The transaction address specifies the message destination while the transaction data specifies the interrupt vector, trigger mode and delivery mode. System software is expected to configure the message data and address during MSI device configuration, allocating one or more no-shared messages to MSI capable devices. Since the MSI address and data are configured through PCI configuration space, to control these physical interrupts the VMM needs to assume ownership of PCI configuration space. This allows the VMM to capture the guest configuration of message address and data for MSI-capable virtual and assigned guest devices.

5.2.3 Examples of Handling of External Interrupts

The following sections illustrate interrupt processing in a VMM (when used to support the external interrupt virtualization requirements).

Guest Setup The VMM sets up the guest to cause a VM exit to the VMM on external interrupts. This is done by setting the "external-interrupt exiting" VM-execution control in the guest controlling-VMCS.

Processor Treatment of External Interrupt Interrupts are automatically masked by hardware in the processor on VM exit by clearing RFLAGS.IF. The exit-reason field in VMCS is set to 1 to indicate an external interrupt as the exit reason. If the VMM is utilizing the acknowledge-on-exit feature (by setting the acknowledge-interrupt-on-exit bit in guest VM-exit control field), the processor acknowledges the interrupt, retrieves the host vector, and saves the interrupt in the exit-interruption-information field (in the VM-exit information region of the VMCS) before transitioning control to the VMM.

Processing of External Interrupts by VMM Upon VM exit, the VMM can determine the exit cause of an external interrupt by checking the exit-reason field (value = 1) in VMCS. If the acknowledge-interrupt-on-exit control is enabled, the VMM can use the saved host vector (in the exit-interruption-information field) to switch to the appropriate interrupt handler. If acknowledge-interrupt-on-exit is not enabled, the VMM may re-enable interrupts (by setting RFLAGS.IF) to allow vectoring of external interrupts through the monitor/host IDT. The following steps may need to be performed by the VMM to process an external interrupt:

- Host Owned I/O Devices: For host-owned I/O devices, the interrupting device is owned by the VMM (or hosting OS in a hosted VMM). In this model, the interrupt service routine in the VMM/host driver is invoked and, upon ISR completion, the appropriate write sequences (TPR updates, EOI etc.) to respective interrupt controllers are performed as normal. If the work completion indicated by the driver implies virtual device activity, the VMM runs the virtual device emulation. Depending on the device class, physical device activity could imply activity by multiple virtual devices mapped over the device. For each affected virtual device, the VMM injects a virtual external interrupt event to respective guest virtual machines. The guest driver interacts with the emulated virtual device to process the virtual interrupt. The interrupt controller emulation in the VMM supports various guest accesses to the VMM's virtual interrupt controller.

- Guest Assigned I/O Devices: For assigned I/O devices, either the VMM uses a software proxy or it can directly map the physical device to the assigned VM. In both cases, servicing of the interrupt condition on the physical device is initiated by the driver running inside the guest VM. With host control of external interrupts, interrupts from assigned physical devices cause VM exits to the VMM and vectoring through the host IDT to the registered VMM interrupt handler. To unblock delivery of other low priority platform interrupts, the VMM interrupt handler must mask the interrupt source (for level triggered interrupts) and issue the appropriate EOI write sequences. Once the physical interrupt source is masked and the platform EOI generated, the VMM can map the host vector to its corresponding guest vector to inject the virtual interrupt into the assigned VM. The guest software does EOI write sequences to its virtual interrupt controller after completing interrupt processing. For level triggered interrupts, these EOI writes to the virtual interrupt controller may be trapped by the VMM which may in turn unmask the previously masked interrupt source.

Generation of Virtual Interrupt Events by VMM The following provides some of the general steps that need to be taken by VMM designs when generating virtual interrupts:

1. Check virtual processor interruptibility state. The virtual processor interruptibility state is reflected in the guest RFLAGS.IF flag and the processor interruptibility-state saved in the guest state area of the controlling-VMCS. If RFLAGS.IF is set and the interruptibility state indicates readiness to take external interrupts (STI-masking and MOV-SS/POP-SS-masking bits are clear), the guest virtual processor is ready to take external interrupts. If the VMM design supports non-active guest sleep states, the VMM needs to make sure the current guest sleep state allows injection of external interrupt events.

2. If the guest virtual processor state is currently not interruptible, a VMM may utilize the "interrupt-window exiting" VM-execution control to notify the VM (through a VM exit) when the virtual processor state changes to interruptible state.

3. Check the virtual interrupt controller state. If the guest VM exposes a virtual local APIC, the current value of its processor priority register specifies if guest software allows dispensing an external virtual interrupt with a specific priority to the virtual processor. If the virtual interrupt is routed through the local vector table (LVT) entry of the local APIC,

the mask bits in the corresponding LVT entry specifies if the interrupt is currently masked. Similarly, the virtual interrupt controller's current mask (IO-APIC or PIC) and priority settings reflect guest state to accept specific external interrupts. The VMM needs to check both the virtual processor and interrupt controller states to verify its guest interruptibility state. If the guest is currently interruptible, the VMM can inject the virtual interrupt. If the current guest state does not allow injecting a virtual interrupt, the interrupt needs to be queued by the VMM until it can be delivered.

4. Prioritize the use of VM-entry event injection. A VMM may use VM-entry event injection to deliver various virtual events (such as external interrupts, exceptions, traps, and so forth). VMM designs may prioritize use of virtualinterrupt injection between these event types. Since each VM entry allows injection of one event, depending on the VMM event priority policies, the VMM may need to queue the external virtual interrupt if a higher priority event is to be delivered on the next VM entry. Since the VMM has masked this particular interrupt source (if it was level triggered) and done EOI to the platform interrupt controller, other platform interrupts can be serviced while this virtual interrupt event is queued for later delivery to the VM.

5. Update the virtual interrupt controller state. When the above checks have passed, before generating the virtual interrupt to the guest, the VMM updates the virtual interrupt controller state (Local-APIC, IO-APIC and/or PIC) to reflect assertion of the virtual interrupt. This involves updating the various interrupt capture registers, and priority registers as done by the respective hardware interrupt controllers. Updating the virtual interrupt controller state is required for proper interrupt event processing by guest software.

6. Inject the virtual interrupt on VM entry. To inject an external virtual interrupt to a guest VM, the VMM sets up the VM-entry interruption-information field in the guest controlling-VMCS before entry to guest using VMRESUME. Upon VM entry, the processor will use this vector to access the gate in guest's IDT and the value of RFLAGS and EIP in guest-state area of controlling-VMCS is pushed on the guest stack. If the guest RFLAGS.IF is clear, the STI-masking bit is set, or the MOV- SS/POP-SS-masking bit is set, the VM entry will fail and the processor will load state from the host-state area of the working VMCS as if a VM exit had occurred.

A APPENDIX: First steps in programming a VMM

The VMM software layer runs at the most privileged level and has complete ownership of the underlying system hardware. The VMM controls creation of a VM, transfers control to a VM, and manages situations that can cause transitions between the guest VMs and host VMM. The VMM allows the VMs to share the underlying hardware and yet provides isolation between the VMs. The guest software executing in a VM is unaware of any transitions that might have occurred between the VM and its host.

A.1 Discovering support for VMX

Before system software enters into VMX operation, it must discover the presence of VMX support in the processor. System software can determine whether a processor supports VMX operation using CPUID. If CPUID.1:ECX.VMX[bit 5] = 1, then VMX operation is supported. See figure 6.

VMX architecture is designed to be extensible so that future processors in VMX operation can support additional features not present in first-generation implementations of the VMX architecture. The availability of extensible VMX features is reported to software using a set of VMX capability MSRs.

A.2 Enabling and entering VMX operation

Before system software can enter VMX operation, it enables VMX by setting CR4.VMXE[bit 13] = 1. VMX operation is then entered by executing the VMXON instruction. VMXON causes an invalid-opcode exception (#UD) if executed with CR4.VMXE = 0. Once in VMX operation, it is not possible to clear CR4.VMXE. System software leaves VMX operation by executing the VMXOFF instruction. CR4.VMXE can be cleared outside of VMX operation after executing of VMXOFF.

A.3 Software Access to the VMCS and related structures

This section details guidelines that software should observe when accessing a VMCS and related structures. It also provides descriptions of consequences for failing to follow guidelines.

A.3.1 Software Access to the Virtual-Machine Control Structure

To ensure proper processor behavior, software should observe certain guidelines when accessing an active VMCS. No VMCS should ever be active on more than one logical processor. If a VMCS is to be "migrated" from one logical processor to another, the first logical processor should execute VMCLEAR for the VMCS (to make it inactive on that logical processor and to ensure that all VMCS data are in memory) before the other logical processor executes VMPTRLD for the VMCS (to make it active on the second logical processor).

Software should never access or modify the VMCS data of an active VMCS using ordinary memory operations, in part because the format used to store the VMCS data is implementation-specific and not architecturally defined, and

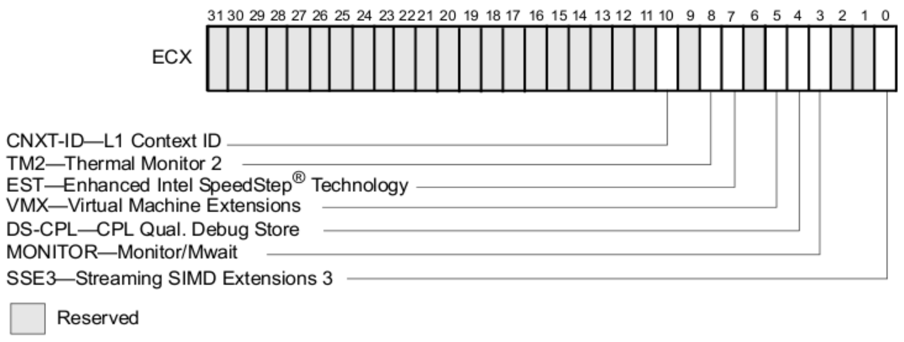


Figure 6: CPUID Extended Feature Information ECX

also because a logical processor may maintain some VMCS data of an active VMCS on the processor and not in the VMCS region. Software can avoid such problems by removing any linear-address mappings to a VMCS region before executing a VMPTRLD for that region and by not remapping it until after executing VMCLEAR for that region. Software should use the VMREAD and VMWRITE instructions to access the different fields in the current VMCS. Software should initialize all fields in a VMCS (using VMWRITE) before using the VMCS for VM entry.

A.3.2 VMREAD, VMWRITE, and Encodings of VMCS Fields

Every field of the VMCS is associated with a 32-bit value that is its encoding. The encoding is provided in an operand to VMREAD and VMWRITE when software wishes to read or write that field. These instructions fail if given, in 64-bit mode, an operand that sets an encoding bit beyond bit 32. The structure of the 32-bit encodings of the VMCS components is determined principally by the width of the fields and their function in the VMCS.

A.3.3 Software Access to Related Structures

In addition to data in the VMCS region itself, VMX non-root operation can be controlled by data structures that are referenced by pointers in a VMCS (for example, the I/O bitmaps). Note that, while the pointers to these data structures are parts of the VMCS, the data structures themselves are not. They are not accessible using VMREAD and VMWRITE but by ordinary memory writes. Software should ensure that each such data structure is modified only when no logical processor with a current VMCS that references it is in VMX non-root operation.

A.3.4 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region) that the logical processor uses to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer is subject to the limitations that apply to VMCS pointers. The amount of memory required for the VMXON region is the same as that required for a VMCS region. This size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC.

Before executing VMXON, software should write the VMCS revision identifier to the VMXON region. It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor.

A.3.5 Using VMCLEAR to initialize a VMCS region

To avoid the uncertainties of implementation-specific behavior, software should execute VMCLEAR on a VMCS region before making the corresponding VMCS active with VMPTRLD. A logical processor uses the VMCS region to maintain the launch state of the corresponding VMCS. The launch state may be clear or launched. The VMCLEAR instruction puts the VMCS referenced by its operand into the clear state. The VMLAUNCH instruction requires a VMCS whose launch state is clear and changes its launch state to launched. The VMRESUME instruction requires a VMCS whose launch state is launched. There are no other ways to modify the launch state of a VMCS (it cannot be modified using VMWRITE) and there is no direct way to read it (it cannot be read using VMREAD). Improper software usage (for example, software writing to the VMCS data of an active VMCS) may leave the launch state undefined. The following software usage is consistent with these limitations:

- VMCLEAR should be executed for a VMCS before it is used for VM entry.
- VMLAUNCH should be used for the first VM entry using a VMCS after VMCLEAR has been executed for that VMCS.
- VMRESUME should be used for any subsequent VM entry using a VMCS (until the next execution of VMCLEAR for the VMCS).

It is expected that, in general, VMRESUME will have lower latency than VMLAUNCH. Since "migrating" a VMCS from one logical processor to another requires use of VMCLEAR, which sets the launch state of the VMCS to "clear," such migration requires the next VM entry to be performed using VMLAUNCH. Software developers can avoid the performance cost of increased VM-entry latency by avoiding unnecessary migration of a VMCS from one logical processor to another.

A.3.6 VMCS states

A VMCS is referred to as a controlling VMCS if it is the current VMCS on a logical processor in VMX non-root operation. A current VMCS for controlling a logical processor in VMX non-root operation may be referred to as a working VMCS if the logical processor is not in VMX non-root operation. The relationship of active, current (i.e. working) and controlling VMCS during VMX operation is shown in Figure 7.

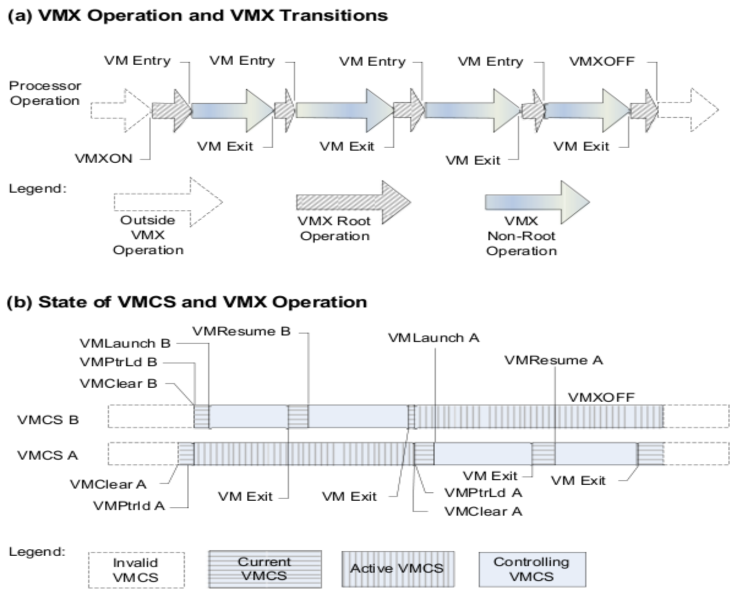


Figure 7: VMX Transitions and States of VMCS in a Logical Processor

A.4 Supporting processor operating modes in guest invironments

Typically, VMMs transfer control to a VM using VMX transitions referred to as VM entries. The boundary conditions that define what a VM is allowed to execute in isolation are specified in a virtual-machine control structure (VMCS). Processors may fix certain bits in CR0 and CR4 to specific values and not support other values. The first processors to support VMX operation require that CR0.PE and CR0.PG be 1 in VMX operation. Thus, a VM entry is allowed only to guests with paging enabled that are in protected mode or in virtual-8086 mode. Guest execution in other processor operating modes need to be specially handled by the VMM. One example of such a condition is guest execution in real-mode. A VMM could support guest real-mode execution using at least two approaches:

- By using a fast instruction set emulator in the VMM.
- By using the similarity between real-mode and virtual-8086 mode to support real-mode guest execution in a virtual-8086 container. The virtual-8086 container may be implemented as a virtual-8086 container task within a monitor that emulates real-mode guest state and instructions, or by running the guest VM as the virtual-8086 container (by entering the guest with RFLAGS.VM1 set). Attempts by real-mode code to access privileged state outside the virtual-8086 container would trap to the VMM and would also need to be emulated.

Another example of such a condition is guest execution in protected mode with paging disabled. A VMM could support such guest execution by using "identity" page tables to emulate unpaged protected mode.

A.4.1 Emulating Guest Execution

In certain conditions, VMMs may resort to using a virtual-8086 container to support guest execution in operating modes not supported by VMX. But for other conditions, VMMs may need to resort to emulating guest execution. These are example conditions that require guest emulation in the VMM:

- Programming conditions that are not allowed by the VMX consistency checks. Examples of this include transient conditions introduced when switching between real-mode and protected mode (where some segment may not be consistent with the operating mode).
- Conditions of guest task switching. Task switches always cause VM exits. To correctly advance the guest state, the monitor needs to emulate the guest task switching behavior.

A.5 Using VMX instructions

Software is required to check RFLAGS.CF and RFLAGS.ZF to determine the success or failure of VMX instruction executions. After a VM-entry instruction (VMRESUME or VMLAUNCH) successfully completes the general checks and checks on VMX controls and the host-state area, any errors encountered while loading of guest-state (due to bad guest-state or bad MSR loading) causes the processor to load state from the host-state area of the working VMCS as if a VM exit had occurred. This failure behavior differs from that of VM exits in that no guest-state is saved to the guest-state area. A VMM can detect its VM-exit handler was invoked by such a failure by checking bit 31 (for 1) in the exit reason field of the working VMCS and further identify the failure by using the exit qualification field.

A.6 VMM setup & tear down

VMMs need to ensure that the processor is running in protected mode with paging before entering VMX operation. The following list describes the minimal steps required to enter VMX root operation with a VMM running at CPL = 0.

- Check VMX support in processor using CPUID.
- Determine the VMX capabilities supported by the processor through the VMX capability MSRs.
- Create a VMXON region in non-pageable memory of a size specified by IA32_VMX_BASIC MSR and aligned to a 4-KByte boundary. Software should read the capability MSRs to determine width of the physical addresses that may be used for the VMXON region and ensure the entire VMXON region can be addressed by addresses with that width. Also, software must ensure that the VMXON region is hosted in cache-coherent memory.
- Initialize the version identifier in the VMXON region (the first 32 bits) with the VMCS revision identifier reported by capability MSRs.

- Ensure the current processor operating mode meets the required CR0 fixed bits (CR0.PE = 1, CR0.PG = 1). Other required CR0 fixed bits can be detected through the IA32_VMX_CR0_FIXED0 and IA32_VMX_CR0_FIXED1 MSRs.

- Enable VMX operation by setting CR4.VMXE = 1.

- Ensure the resultant CR4 value supports all the CR4 fixed bits reported in the IA32_VMX_CR4_FIXED0 and IA32_VMX_CR4_FIXED1 MSRs. Ensure that the IA32_FEATURE_CONTROL MSR (MSR index 3AH) has been properly programmed and that its lock bit is set (Bit 0 = 1). This MSR is generally configured by the BIOS using WRMSR.

- Execute VMXON with the physical address of the VMXON region as the operand. Check successful execution of VMXON by checking if RFLAGS.CF = 0.

Upon successful execution of the steps above, the processor is in VMX root operation. A VMM executing in VMX root operation and CPL = 0 leaves VMX operation by executing VMXOFF and verifies successful execution by checking if RFLAGS.CF = 0 and RFLAGS.ZF = 0.

A.7 Preparation and launching a virtual machine

The following list describes the minimal steps required by the VMM to set up and launch a guest VM.

- Create a VMCS region in non-pageable memory of size specified by the VMX capability MSR IA32_VMX_BASIC and aligned to 4-KBytes. Software should read the capability MSRs to determine width of the physical addresses that may be used for a VMCS region and ensure the entire VMCS region can be addressed by addresses with that width. The term "guest-VMCS address" refers to the physical address of the new VMCS region for the following steps.

- Initialize the version identifier in the VMCS (first 32 bits) with the VMCS revision identifier reported by the VMX capability MSR IA32_VMX_BASIC.

- Execute the VMCLEAR instruction by supplying the guest-VMCS address. This will initialize the new VMCS region in memory and set the launch state of the VMCS to "clear". This action also invalidates the working-VMCS pointer register to FFFFFFFF_FFFFFFFFH. Software should verify successful execution of VMCLEAR by checking if RFLAGS.CF = 0 and RFLAGS.ZF = 0.

- Execute the VMPTRLD instruction by supplying the guest-VMCS address. This initializes the working-VMCS pointer with the new VMCS region's physical address.

- Issue a sequence of VMWRITEs to initialize various host-state area fields in the working VMCS. The initialization sets up the context and entry-points to the VMM upon subsequent VM exits from the guest. Host-state fields include control registers (CR0, CR3 and CR4), selector fields for the segment registers (CS, SS, DS, ES, FS, GS and TR), and base-address fields (for FS, GS, TR, GDTR and IDTR; RSP, RIP and the MSRs that control fast system calls).

- Use VMWRITEs to set up the various VM-exit control fields, VM-entry control fields, and VM-execution control fields in the VMCS. Care should be taken to make sure the settings of individual fields match the allowed 0 and 1 settings for the respective controls as reported by the VMX capability MSRs. Any settings inconsistent with the settings reported by the capability MSRs will cause VM entries to fail.

- Use VMWRITE to initialize various guest-state area fields in the working VMCS. This sets up the context and entry-point for guest execution upon VM entry.

- The VMM is required to set up guest-state that complies with these consistency checks:

  - If the VMM design requires the initial VM launch to cause guest software (typically the guest virtual BIOS) execution from the guest's reset vector, it may need to initialize the guest execution state to reflect the state of a physical processor at power-on reset.
  - The VMM may need to initialize additional guest execution state that is not captured in the VMCS guest-state area by loading them directly on the respective processor registers. Examples include general purpose registers, the CR2 control register, debug registers, floating point registers and so forth. VMM may support lazy loading of FPU, MMX, SSE, and SSE2 states with CR0.TS = 1.

- Execute VMLAUNCH to launch the guest VM. If VMLAUNCH fails due to any consistency checks before guest-state loading, RFLAGS.CF or RFLAGS.ZF will be set and the VM-instruction error field will contain the errorcode. If guest-state consistency checks fail upon guest-state loading, the processor loads state from the host-state area as if a VM exit had occurred.

VMLAUNCH updates the controlling-VMCS pointer with the working-VMCS pointer and saves the old value of controlling-VMCS as the parent pointer. In addition, the launch state of the guest VMCS is changed to "launched" from "clear". Any programmed exit conditions will cause the guest to VM exit to the VMM. The VMM should execute VMRESUME instruction for subsequent VM entries to guests in a "launched" state.

A.8 Handling of VM exits

This section provides examples of software steps involved in a VMM's handling of VMexit conditions:

- Determine the exit reason through a VMREAD of the exit-reason field in the working-VMCS.

- VMREAD the exit-qualification from the VMCS if the exit-reason field provides a valid qualification. The exit-qualification field provides additional details on the VM-exit condition. For example, in case of page faults, the exit-qualification field provides the guest linear address that caused the page fault.
- Depending on the exit reason, fetch other relevant fields from the VMCS.
- Handle the VM-exit condition appropriately in the VMM. This may involve the VMM emulating one or more guest instructions, programming the underlying host hardware resources, and then re-entering the VM to continue execution.

A.8.1 Handling VM Exits Due to Exceptions

As noted before, an exception causes a VM exit if the bit corresponding to the exception's vector is set in the exception bitmap. (For page faults, the error code also determines whether a VM exit occurs.) This section provide some guidelines of how a VMM might handle such exceptions. Exceptions result when a logical processor encounters an unusual condition that software may not have expected. When guest software encounters an exception, it may be the case that the condition was caused by the guest software. For example, a guest application may attempt to access a page that is restricted to supervisor access. Alternatively, the condition causing the exception may have been established by the VMM. For example, a guest OS may attempt to access a page that the VMM has chosen to make not present. When the condition causing an exception was established by guest software, the VMM may choose to reflect the exception to guest software. When the condition was established by the VMM itself, the VMM may choose to resume guest software after removing the condition.

Reflecting Exceptions to Guest Software If the VMM determines that a VM exit was caused by an exception due to a condition established by guest software, it may reflect that exception to guest software. The VMM would cause the exception to be delivered to guest software, where it can be handled as it would be if the guest were running on a physical machine. This section describes how that may be done. In general, the VMM can deliver the exception to guest software using VM-entry event injection as described before. The VMM can copy (using VMREAD and VMWRITE) the contents of the VM-exit interruption-information field (which is valid, since the VM exit was caused by an exception) to the VM-entry interruption-information field (which, if valid, will cause the exception to be delivered as part of the next VM entry). The VMM would also copy the contents of the VM-exit interruption errorcode field to the VM-entry exception error-code field; this need not be done if bit 11 (error code valid) is clear in the VM-exit interruption-information field. After this, the VMM can execute VMRESUME.

Resuming Guest Software after Handling an Exception If the VMM determines that a VM exit was caused by an exception due to a condition established by the VMM itself, it may choose to resume guest software after removing the condition. The approach for removing the condition may be specific to the

VMM's software architecture. and algorithms This section describes how guest software may be resumed after removing the condition. In general, the VMM can resume guest software simply by executing VMRESUME. The following items provide details of cases that may require special handling:

A.9 Multiprocessor considerations

The most common VMM design will be the symmetric VMM. This type of VMM runs the same VMM binary on all logical processors. Like a symmetric operating system, the symmetric VMM is written to ensure all critical data is updated by only one processor at a time, IO devices are accessed sequentially, and so forth. Asymmetric VMM designs are possible. For example, an asymmetric VMM may run its scheduler on one processor and run just enough of the VMM on other processors to allow the correct execution of guest VMs. The remainder of this section focuses on the multi-processor considerations for a symmetric VMM.

A symmetric VMM design does not preclude asymmetry in its operations. For example, a symmetric VMM can support asymmetric allocation of logical processor resources to guests. Multiple logical processors can be brought into a single guest environment to support an MP-aware guest OS. Because an active VMCS can not control more than one logical processor simultaneously, a symmetric VMM must make copies of its VMCS to control the VM allocated to support an MP-aware guest OS. Care must be taken when accessing data structures shared between these VMCSs.

Although it may be easier to develop a VMM that assumes a fully-symmetric view of hardware capabilities (with all processors supporting the same processor feature sets, including the same revision of VMX), there are advantages in developing a VMM that comprehends different levels of VMX capability (reported by VMX capability MSRs). One possible advantage of such an approach could be that an existing software installation (VMM and guest software stack) could continue to run without requiring software upgrades to the VMM, when the software installation is upgraded to run on hardware with enhancements in the processor's VMX capabilities. Another advantage could be that a single software installation image, consisting of a VMM and guests, could be deployed to multiple hardware platforms with varying VMX capabilities. In such cases, the VMM could fall back to a common subset of VMX features supported by all VMX revisions, or choose to understand the asymmetry of the VMX capabilities and assign VMs accordingly. This section outlines some of the considerations to keep in mind when developing an MP-aware VMM.

A.9.1 Initialization

Before enabling VMX, an MP-aware VMM must check to make sure that all processors in the system are compatible and support features required. This can be done by:

- Checking the CPUID on each logical processor to ensure VMX is supported and that the overall feature set of each logical processor is compatible.
- Checking VMCS revision identifiers on each logical processor.

- Checking each of the “allowed-1” or “allowed-0” fields of the VMX capability MSR’s on each processor.

A.9.2 Moving a VMCS Between Processors

An MP-aware VMM is free to assign any logical processor to a VM. But for performance considerations, moving a guest VMCS to another logical processor is slower than resuming that guest VMCS on the same logical processor. Certain VMX performance features (such as caching of portions of the VMCS in the processor) are optimized for a guest VMCS that runs on the same logical processor. The reasons are:

- To restart a guest on the same logical processor, a VMM can use VMRESUME. VMRESUME is expected to be faster than VMLAUNCH in general.
- To migrate a VMCS to another logical processor, a VMM must use the sequence of VMCLEAR, VMPTRLD and VMLAUNCH.
- Operations involving VMCLEAR can impact performance negatively.

A VMM scheduler should make an effort to schedule a guest VMCS to run on the logical processor where it last ran. Such a scheduler might also benefit from doing lazy VMCLEARs (that is: performing a VMCLEAR on a VMCS only when the scheduler knows the VMCS is being moved to a new logical processor).

The remainder of this section describes the steps a VMM must take to move a VMCS from one processor to another. A VMM must check the VMCS revision identifier in the VMX capability MSR IA32_VMX_BASIC to determine if the VMCS regions are identical between all logical processors. If the VMCS regions are identical (same revision ID) the following sequence can be used to move or copy the VMCS from one logical processor to another:

- Perform a VMCLEAR operation on the source logical processor. This ensures that all VMCS data that may be cached by the processor are flushed to memory.
- Copy the VMCS region from one memory location to another location. This is an optional step assuming the VMM wishes to relocate the VMCS or move the VMCS to another system.
- Perform a VMPTRLD of the physical address of VMCS region on the destination processor to establish its current VMCS pointer.

If the revision identifiers are different, each field must be copied to an intermediate structure using individual reads (VMREAD) from the source fields and writes (VMWRITE) to destination fields. Care must be taken on fields that are hard-wired to certain values on some processor implementations.

A.10 Performance considerations

VMX provides hardware features that may be used for improving processor virtualization performance. VMMs must be designed to use this support properly. The basic idea behind most of these performance optimizations of the VMM is to reduce the number of VM exits while executing a guest VM. This section lists ways that VMMs can take advantage of the performance enhancing features in VMX.

- Read Access to Control Registers. Analysis of common client workloads with common PC operating systems in a virtual machine shows a large number of VM-exits are caused by control register read accesses (particularly CR0). Reads of CR0 and CR4 does not cause VM exits. Instead, they return values from the CR0/CR4 read-shadows configured by the VMM in the guest controlling-VMCS with the guest-expected values.

- Write Access to Control Registers. Most VMM designs require only certain bits of the control registers to be protected from direct guest access. Write access to CR0/CR4 registers can be reduced by defining the host-owned and guest-owned bits in them through the CR0/CR4 host/guest masks in the VMCS. CR0/CR4 write values by the guest are qualified with the mask bits. If they change only guestowned bits, they are allowed without causing VM exits. Any write that cause changes to host-owned bits cause VM exits and need to be handled by the VMM.

- Access Rights based Page Table protection. For VMM that implement access-rights-based page table protection, the VMCS provides a CR3 target value list that can be consulted by the processor to determine if a VM exit is required. Loading of CR3 with a value matching an entry in the CR3 target-list are allowed to proceed without VM exits. The VMM can utilize the CR3 target-list to save page-table hierarchies whose state is previously verified by the VMM.

- Page-fault handling. Another common cause for a VM exit is due to page-faults induced by guest address remapping done through virtual memory virtualization. VMX provides page-fault error-code mask and match fields in the VMCS to filter VM exits due to page-faults based on their cause (reflected in the error-code).

References

[1] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manuals* http://www.intel.com/products/processor/manuals/

[2] Intel Corp. *Intel Virtualization Technology*

[3] Tom Shanley *Protected Mode Software Architecture* Addison-Wesley.