

Lecture 03 Intel VMX (part 2): Chapters 28-30

A design-oriented guide to Chapters 28 through 30 of the Intel 64 and IA-32 Architectures Software Developer Software Developer Manual, Volume 3. Order Number 325384-092US, June 2026.

Contents

- 01** The runtime loop
loop
- 02** Native execution with escape hatches
- 03** Choosing where to intercept
- 04** Interrupts, NMIs, and hidden boundaries
- 05** Non-root facilities
- 06** VM entry is a validated transaction
- 07** VM exit is transfer plus explanation
- 08** Round trips and design patterns

01

The runtime loop

Three questions the runtime chapters answer

Chapters 26 and 27 describe VMX operation, VMCS regions, state fields, and controls. Chapters 28 through 30 make that machinery move.

-
- 1** While the guest is running, which operations execute natively and which return control to the hypervisor?
 - 2** How does the processor safely construct a guest context during VM entry?
 - 3** How does it stop the guest, explain what happened, and construct a safe host context during VM exit?
-

The useful mental model is not "the CPU switches into an emulator." The guest normally runs directly on the processor. VMX adds policy checks at carefully chosen architectural boundaries.

The center of hardware virtualization



no exit condition → keep executing guest instructions natively
exit condition → VM exit → host handles cause → re-enter

Performance depends on keeping the native-execution portion wide. Correctness depends on making each boundary precise.

02

Native execution with escape hatches

Six possible outcomes for a guest operation

Most guest instructions use the ordinary execution engine, caches, branch predictors, page walkers, and arithmetic units. Selected instructions and events are routed through VMX policy instead.

NATIVE Execute with normal architectural behavior.	MODIFIED Execute with VMX-modified behavior.	FAULT Cause a fault inside the guest.
FAULT-LIKE EXIT Cause a VM exit before it executes.	TRAP-LIKE EXIT Complete and then cause a VM exit.	VIRTUALIZED Be handled by a dedicated facility without an exit.

■ **Design reading.** That range is deliberate. "Always trap" would be simple but slow; "always execute" would be fast but unsafe. VMX turns individual architectural surfaces into independently selectable boundaries.

Fault-like and trap-like VM exits

FAULT-LIKE · THE COMMON CASE

```
guest reaches intercepted instruction
→ VM exit before it executes
→ saved guest RIP points at it
→ VMM emulates, rejects, or
  changes policy
```

If `CPUID` exits, the guest has not yet received results. The VMM synthesizes values in the guest register frame, advances guest RIP by the reported instruction length, and resumes.

TRAP-LIKE

The triggering operation completes before the exit. Saved state reflects its effects, and guest RIP normally identifies the following instruction.

Examples: certain debug conditions, APIC-write emulation, bus-lock detection, and some virtualized control-register or x2APIC operations.

A handler must know whether it is completing an operation that never ran or reacting to one that already changed architectural state.

Why use trap-like exits outside debugging?

A non-debug trap-like exit is useful when the guest operation is safe to complete, but its result creates a condition the VMM must handle.

VMM REQUIREMENT
Decide whether an operation is allowed, reject it, or emulate it with different results
Observe the completed result, respond to a newly reached threshold, or handle a consequence detected during execution

APPROPRIATE BOUNDARY
Fault-like exit before execution
Trap-like exit after execution

The second pattern avoids making the VMM reproduce work hardware can already do correctly. Hardware performs the common operation, then exits only when its result crosses a boundary the VMM selected.

Example: CR8 and interrupt priority

On x86-64, CR8 represents the task-priority class of the local APIC. An APIC vector's high four bits define its priority class.

```
priority class = vector >> 4
```

```
vector 0x52 → class 5
```

```
vector 0x91 → class 9
```

Ignoring other blocking conditions, an interrupt is eligible only when its class exceeds the processor's current priority. With CR8 = 8, classes 0 through 8 are blocked and 9 through 15 may be delivered.

ONE INPUT, NOT THE WHOLE ANSWER

The APIC's effective Processor Priority Register also accounts for the highest-priority interrupt already in service. CR8 is one input to the final decision rather than an interrupt queue by itself.

A TPR-threshold exit fires after the write

```
virtual CR8 = 8
pending virtual interrupt =
    vector 0x52, class 5

guest executes MOV CR8, 0
→ hardware records virtual CR8 = 0
→ class-5 interrupt becomes eligible
→ TPR-threshold condition true
→ VM exit occurs after the MOV
```

This ordering is exactly what the VMM wants. Hardware has already updated the virtual task priority; the VMM only needs an opportunity to notice that a pending virtual interrupt may now be delivered. On the exit, saved guest state contains the new priority and guest RIP is past the MOV. Replaying it would be incorrect.

TPR shadowing and related APIC-virtualization controls make the common path cheaper still: reads and writes update virtual APIC state without exiting, and a threshold exit asks for software only when lowering priority creates relevant work.

What Linux actually does with priority

During local-APIC initialization, Linux sets the Task Priority Register to 0x10, equivalent to CR8 = 1 , so APIC vectors 0x10 through 0x1F cannot be mistaken for processor exceptions. It does not subsequently change this value.

For ordinary critical sections, native x86 Linux primarily clears and restores RFLAGS.IF with CLI and STI , blocking blocking all maskable interrupts rather than dynamically raising and lowering CR8.

0xFD	0xFC · 0xFB	0xEC	0x80
Rescheduling IPIs	Cross-CPU function calls	Local timer	Software INT : not filtered by CR8

The high nibble is an APIC delivery priority, not a priority attached to every use of an IDT vector. Software interrupts and synchronous processor exceptions are not filtered by CR8.

Example: a LOCK prefix is not a bus lock

For an atomic read-modify-write whose operand lies entirely within one cache line in ordinary write-back memory, a modern processor normally obtains exclusive ownership through cache coherence.

```
one cacheable line
→ request exclusive ownership
→ invalidate competing copies
→ perform the atomic update
→ unrelated traffic continues
```

Only *relatively* cheap: a contended atomic can still bounce the line the line between cores, and false sharing makes unrelated data data contend.

WHEN A REAL BUS LOCK IS REQUIRED

When cache coherence cannot represent the whole atomic operation, for example a locked operand crossing a cache-line boundary or a memory type for which cache locking cannot be used.

The processor then serializes more broadly through the memory interconnect. That can delay other cores and memory agents accessing unrelated cache lines, making repeated bus locks a possible denial-of-service technique.

Why bus-lock detection must be trap-like

Whether the instruction requires a bus lock may depend on information discovered as it executes:

The translated physical address	Alignment and whether it spans cache lines	The effective memory type	The available atomicity mechanism for that access
---------------------------------	--	---------------------------	---

The processor completes the atomic instruction, reports that it caused a bus lock, and then exits. The VMM cannot undo that transaction, but it can log the event, charge or throttle the vCPU, or terminate a guest that repeats it. The handler must resume *after* the instruction; replaying it would perform the atomic update, and potentially the bus lock, a second time.

- Not to be confused with split-lock detection, a separate policy that can raise an alignment-check exception. The former reports an operation after it caused a bus lock; the latter is fault-oriented.

A fault-like exit lets the VMM decide what the operation should do. A trap-like exit lets hardware complete an operation whose result or runtime consequence tells the VMM that software attention is now required.

Exception priority comes before virtualization policy

An instruction can be both architecturally invalid and configured to cause a VM exit. VMX defines priority between ordinary faults and exits.

RDMSR at CPL 3 generates `#GP`. It does not give an unprivileged guest program a route to the hypervisor merely because MSR exiting is enabled.

At CPL 0, an intercepted RDMSR of an unimplemented index can exit before the processor generates the normal `#GP` for that index.

Operand-fetch faults may occur before an exit whose decision depends on the operand.

WHY THIS MAKES SENSE

Virtualization should preserve the guest's architectural protection model. Otherwise, configuring an interception could make an instruction behave observably differently even before the hypervisor chose to emulate it. Priority rules define exactly which boundary the VMM inherits.

03

Choosing where to intercept

Instructions that always exit

CATEGORY	EXAMPLES	WHY UNCONDITIONAL CONTROL IS USEFUL
Machine identity	CPUID	Present a stable virtual CPU model rather than the raw host model
Global cache or mode behavior	INVD, XSETBV	Direct execution can affect machine-wide behavior or host-managed extended state
Secure-launch machinery	GETSEC	The guest must not silently alter the platform's real measured-launch state
VMX ownership	VMXON, VMLAUNCH, VMRESUME	A guest must not acquire or manipulate the outer hypervisor's real VMX control structures
Nested-virtualization interface interface	VMREAD, VMWRITE	L0 must emulate or selectively accelerate the VMX machine that L1 believes it owns

VMCS shadowing is the exception to the simple nested story: selected guest VMREAD and VMWRITE operations can access a shadow access a shadow VMCS without exiting. The shadow still cannot drive real VM entry; L0 retains that authority.

Conditional exits are the fast-path policy

MECHANISM	EXAMPLES	POLICY GRANULARITY
Single control bit	HLT, INVLPG, RDPMC, RDTSC, MOV DR	All uses of one instruction class
Dense bitmap	I/O, RDMSR, WRMSR, ENCLS, PCONFIG	One port, MSR index, or leaf at a time
Bit ownership plus shadow	MOV CR0, MOV CR4, CLTS, LMSW	Individual control-register bits
Exact-value targets	MOV to CR3	A small set of hot values bypasses a general exit rule
Timing threshold	PAUSE-loop exiting, instruction timeout	Exit only after behavior looks costly
VMREAD/VMWRITE bitmap	Nested-hypervisor VMCS accesses	Individual VMCS field encodings

■ **Design reading.** "Is this port intercepted?" is a bit lookup. "Does this write cross host-owned CR4 bits?" is a masked comparison. "Is comparison. "Is this one of a few permitted CR3 roots?" is a small exact match. VMX avoids a general rule language because because predictable hardware lookup is faster and easier to validate.

Controls can modify an instruction instead of trapping it

It is tempting to think an execution control only chooses between "native" and "VM exit." Some controls instead give the instruction a virtualized meaning.

A masked `MOV to CR0 or CR4` may update guest-owned bits while leaving host-owned bits unchanged

TSC offsetting and scaling transform timestamp reads in hardware

APIC virtualization changes the behavior of APIC and x2APIC accesses

`IA32_TSC_DEADLINE` writes can program a guest timer rather than than the physical APIC timer

`IA32_SPEC_CTRL` virtualization combines guest-visible state host-enforced bits

`VMFUNC` can perform an enabled VM function without exiting

```
ordinary native execution
hardware-virtualized execution
VM exit for software policy
```

The middle path is how later processor generations remove exits from operations that were previously expensive to virtualize.

Events, not just instructions, cause exits

Exceptions selected by the exception bitmap	INIT and qualifying SIPI events
Page-fault classes selected by mask and match	Attempted task switches
External interrupts, when exiting is enabled	Expiration of the VMX-preemption timer
NMIs, when NMI exiting is enabled	Bus locks, when VMM detection is enabled
Instruction timeout	Interrupt-window and NMI-window conditions

INTERCEPTION

Something happened that the VMM wants to own, such as a selected exception or physical interrupt.

NOTIFICATION

A condition the VMM was waiting for became true, such as "the guest can finally accept an interrupt." Window exits turn a temporal question into a hardware callback.

Page faults get their own filter

One bitmap bit is often too coarse. Hardware compares the page-fault error code against VMCS mask and match fields:

$$(PFEC \ \& \ PFEC_MASK) == PFEC_MATCH$$

If the comparison matches, exception-bitmap bit 14 has its ordinary meaning. If it does not match, the meaning is reversed.

A shadow-paging VMM can therefore intercept the page-fault classes needed to maintain shadow mappings while allowing ordinary guest page faults to reach the guest OS.

04

Interrupts, NMIs, and hidden boundaries

Interrupt-window exiting: wait without single-stepping

A virtual interrupt is pending but the guest has `RFLAGS.IF=0`. The VMM cannot legally inject it, and exiting after every guest instruction to check `IF` would be disastrous.

```
virtual interrupt pending
→ guest blocks maskable interrupts
→ enable interrupt-window exiting
→ let guest run
→ guest reaches IF=1 with no
  STI/MOV-SS blocking
→ exit before the next instruction
```

The VMM disables window exiting, puts the virtual interrupt in the VM-entry injection fields, and resumes. NMI-window exiting applies the same idea to virtual-NMI blocking.

Intel SDM Vol. 3 · Chapters 28–30

WHY THIS MAKES SENSE

Hardware already evaluates interruptibility at every boundary. Asking it to notify the VMM when the condition changes is far cheaper than reproducing that observation in software.

When external-interrupt exiting is enabled, guest `RFLAGS.IF` does not prevent an external interrupt from causing a VM exit. A guest masking its own virtual interrupts must not be able to starve the host's physical interrupt handling.

From a physical interrupt to the host handler

With external-interrupt exiting enabled, an eligible physical interrupt causes an asynchronous VM exit. No guest instruction caused it, so the handler does not advance guest RIP ; resumption continues at the precise boundary captured by hardware.

The processor saves VMCS-managed guest state and loads VMCS host state, beginning at HOST_RIP with host RFLAGS.IF=0 . The entry stub can therefore save the guest general-purpose registers before another maskable interrupt can preempt it.

The **acknowledge interrupt on exit** control chooses how the physical interrupt reaches host software. "Acknowledge" means accepting the request from the interrupt controller and obtaining its vector, not completing service of the device.

Two ways the host obtains the vector

CONTROL	WHAT VM EXIT DOES	HOW THE HOST OBTAINS THE VECTOR
0	Leaves the interrupt pending in the local APIC and marks exiting-event information invalid	The VMM eventually enables host interrupts and the APIC delivers normally through the host IDT
1	Acknowledges the local APIC during exit, making the request no longer pending, and records the vector	The handler reads the vector and explicitly dispatches the corresponding host interrupt path

```
physical interrupt arrives
→ VM exit acknowledges local APIC
→ VMCS reports vector
→ stub saves guest GPRs with IF=0
→ VMM dispatches host handler
→ handler services device, issues EOI
→ resume without advancing RIP
```

The APIC acknowledgement does not acknowledge the device. A level-triggered device may keep asserting its line until its driver services the source, and the local APIC still needs the normal end-of-interrupt operation.

A physical interrupt is also not automatically a guest interrupt. A host NIC arriving as vector 0xE1 may lead the VMM to separately raise the guest vNIC's vector, perhaps the device belongs only to the host, no guest interrupt is created.

NMI exits use host dispatch, not APIC acknowledgement

The acknowledge-on-exit control does not apply to NMIs. An NMI has fixed vector 2, is not filtered by RFLAGS .IF or CR8 , and does not use the local-APIC in-service and EOI protocol.

physical NMI

- VM exit to HOST_RIP
- save guest GPRs
- recognize event type NMI, vector 2
- dispatch host NMI handler
- complete NMI return/unblocking
- VMRESUME

Hardware loads HOST_RIP ; it does not automatically host IDT vector 2. Exiting -event information identifies NMI and vector 2, after which the VMM must explicitly the host's NMI machinery.

For a small bare-metal VMM, the VM-exit routine may be the complete NMI handler. A hosted hypervisor such as KVM instead connects the exit to the host OS's NMI path so watchdog, profiling, and diagnostic NMIs retain their meaning.

A direct NMI exit establishes real NMI blocking after the exit completes. There is no APIC EOI. A physical host NMI should not ordinarily be injected into the guest.

Injecting an interrupt or NMI on VM entry

The VMM does not call a guest handler or replace guest RIP .

It writes the VM-entry injected-event field and executes VMLAUNCH or VMRESUME . Hardware then performs architectural event delivery using the guest IDT or, when enabled, FRED.

COMPONENT	VALUE
Vector, bits 7:0	2
Event type, bits 10:8	2
Deliver-error-code, bit 11	0
Valid, bit 31	1

With reserved bits clear, that is 0x80000202.

On successful entry, hardware consults guest IDT vector 2, performs any required privilege and stack transition, pushes a return frame containing the current guest RIP , and begins the guest NMI handler. The instruction-length field does not advance RIP for an asynchronous NMI.

NO IMMEDIATE RE-INTERCEPT

NMI exiting does not intercept an NMI the VMM explicitly injects. injects. Otherwise an injected NMI would immediately exit again again and never reach the guest. A problem encountered *during during delivery* can still exit; original-event information preserves preserves the injected NMI.

The STI interrupt shadow

When STI changes RFLAGS.IF from 0 to 1, IF becomes visible as 1 but maskable interrupts are not recognized at the immediately following instruction boundary.

before STI:	IF=0, shadow=0
STI completes:	IF=1, shadow=1
next instruction executes:	IF=1, shadow=1
next instruction retires:	IF=1, shadow=0

The CPU tracks these as hidden one-instruction state and exposes them in the VMCS, so a VM exit can pause the guest between the instruction that creates a shadow and the one that consumes it.

WHY THE CLASSIC IDLE SEQUENCE IS SAFE

```
sti
hlt
```

Without the shadow, an interrupt could be handled between STI and HLT ; the CPU could then halt after consuming the only wakeup event. With the shadow, HLT executes before the pending interrupt is recognized, and the interrupt wakes the halted processor.

The MOV SS shadow, and why shadows survive an exit

Older software commonly changed stacks with two instructions:

```
mov ss, ax
mov esp, new_stack_top
```

An interrupt between them would observe the new stack segment with the old stack pointer. The processor sets a hidden latch after successfully loading SS and clears it after the following instruction executes. It does not inspect whether that instruction actually updates the stack pointer.

LSS, which loads SS and the stack pointer together, avoids the two-instruction inconsistency and does not create the same shadow.

RETIREMENT BOUNDARIES, NOT A TIMER

```
STI completes, blocking established
→ intercepted CUID is next
→ fault-like VM exit before CUID
→ saved guest RIP still = CUID
→ VMCS saves blocking-by-STI = 1
```

The host can run for an arbitrary time without expiring the shadow. On resumption, CUID is still the protected next guest instruction.

What if the VMM injects an NMI while the guest is blocked?

The processor does not silently queue an invalid injection until the guest becomes ready. VM entry checks the request against guest interruptibility state. With virtual NMIs enabled, an NMI injection is incompatible with blocking by STI, blocking by MOV SS, and virtual-NMI blocking from an earlier virtual NMI whose handler has not returned.

No guest instruction and no guest NMI handler executes.

The processor reports a VM-entry failure due to invalid guest state: basic reason 33 with the VM-entry-failure indicator set. Injecting an NMI while STI blocking is recorded has the defined exit-qualification value 3.

THE CORRECT SEQUENCE

```
mark virtual NMI pending in VMM state
→ do not set a valid injection yet
→ enable NMI-window exiting
→ resume guest
→ guest clears STI/MOV-SS or
  virtual-NMI blocking
→ NMI-window VM exit
→ disable NMI-window exiting
→ program injected NMI, 0x80000202
→ resume; hardware enters guest IDT[2]
```

Three interruptibility bits, three boundaries

STATE	ESTABLISHED BY	ENDS AFTER
Blocking by STI	STI enabling interrupts	The following instruction executes or incurs an exception
Blocking by MOV SS	Successful MOV SS or POP SS	The following instruction executes or incurs an exception
Virtual-NMI blocking	Delivery of a virtual NMI	Guest IRET

With virtual NMIs enabled, hardware sets virtual-NMI blocking after injecting the NMI, and the guest's IRET clears it.

Direct versus indirect event exits

DIRECT

The event itself is intercepted: the exception bitmap selects `#PF`, or NMI exiting selects an arriving NMI.

Because the event has not been delivered to the guest, its normal architectural side effects generally have not occurred. A directly exiting page fault does not update guest CR2 ; the relevant address is reported through VM-exit information.

INDIRECT

The original event is allowed, but something encountered while trying to deliver it exits:

- An EPT violation while pushing the guest's interrupt frame
- A nested exception selected by the exception bitmap
- An APIC-access exit
- A task-switch attempt through an IDT task gate
- A triple fault reached during nested delivery

Part of the delivery may already have happened. The processor records both the new exit cause and the original event being delivered.

"An interrupt caused the exit" is insufficient. The VMM must know whether the interrupt was never delivered, was partially delivered, or encountered another event. Otherwise it could lose the original event or inject it twice.

05

Non-root facilities

Task switches always leave non-root operation

Hardware task switching is not allowed to complete in VMX non-root operation. Attempts cause VM exits after the architecturally higher-priority checks described by Chapter 28.

That does not mean the VMM can ignore task-switch semantics. The exit qualification identifies the source of the attempted switch, and faults encountered while resolving task gates or TSS descriptors may take priority.

DESIGN READING

Hardware task switching mutates a large, memory-resident set of state: TSS images, descriptors, busy bits, TR, RFLAGS.NT, and potentially privilege state. Making that entire compound transaction independently virtualizable would greatly complicate precise exit boundaries. VMX instead gives the VMM control before the switch commits.

CET shadow stacks expose a non-rollbackable boundary

A privilege-changing event or far call may switch supervisor shadow stacks. CET first atomically marks the new stack's token busy, then may push return information to it.

If a later push causes a VM exit, the processor can restore the old active shadow stack, but the memory token in the new stack may remain busy. Intel calls this a **prematurely busy shadow stack**. An optional VM-exit control asks hardware to report that condition in exit information.

WHY THIS FIELD EXISTS

A transition can cross a point where an architecturally visible memory update cannot simply be rolled back. The VMM needs an explicit diagnostic bit so it can distinguish that partial transition from an ordinary EPT or event-delivery exit and repair or emulate the state correctly.

The VMX-preemption timer is a non-cooperative deadline

When enabled, VM entry loads a per-vCPU countdown value. It decrements in non-root operation at a capability-reported rate derived from the TSC. Reaching zero causes a VM exit, subject to defined priority and activity-state rules.

```
guest masks interrupts or spins forever
→ preemption timer continues
→ countdown reaches zero
→ VM exit returns control to scheduler
```

It is not the guest's clock and it is not an APIC timer. It is a VMM-owned bound on how long a vCPU may keep the physical processor.

If the "save VMX-preemption timer value" exit control is enabled, an unrelated exit preserves the remaining count in the VMCS. The VMM can then resume the same scheduling budget rather than accidentally granting a new full slice.

Monitor Trap Flag: stepping without guest TF

MTF requests a VM exit at a defined instruction boundary after entry. It is useful for hypervisor debugging and for temporarily allowing one guest operation before regaining control.

GUEST TRAP FLAG	MONITOR TRAP FLAG
Produces guest-visible debug behavior	Produces a VM exit
Interacts with guest debug state and #DB	Belongs to VMM execution policy
Guest can normally observe or manipulate it	Not exposed as guest architectural state

The detailed rules cover event injection, faults, REP iterations, transactional execution, and inactive states, because "after one instruction" is not always one simple boundary on x86.

EPT makes guest-physical an intermediate namespace

```
guest linear address
  → guest page tables
guest-physical address
  → EPT
machine physical address
```

Chapter 28 introduces the runtime role; Chapter 31 specifies the detailed translation mechanism.

This second translation gives the guest realistic control over its own "physical" memory while the VMM retains ownership of actual machine pages.

EPT violation — the requested access is inconsistent with current EPT permissions or mapping

EPT misconfiguration — the EPT structures contain an invalid encoding; these always exit

Treating guest physical memory as a first-class intermediate address space removes the need to rewrite or shadow every guest page-table change.

Two ways to skip the round trip

VMFUNC · PRIVILEGED OPERATIONS WITHOUT EXITS

VMFUNC lets non-root software invoke a function the VMM explicitly enabled. EAX selects the function; the architectural example is EPTP switching, where ECX selects an entry from EPTP list. If index and pointer are valid, hardware switches the the active EPT pointer without a VM exit.

```
VMM enables VM functions globally
→ enables a particular function number
→ supplies bounded function-specific data
→ guest invokes only that operation
```

Some operations are safe only within a menu prepared by the VMM. A table-indexed hardware operation accelerates without giving the guest arbitrary control.

#VE · EPT HANDLING INSIDE THE GUEST

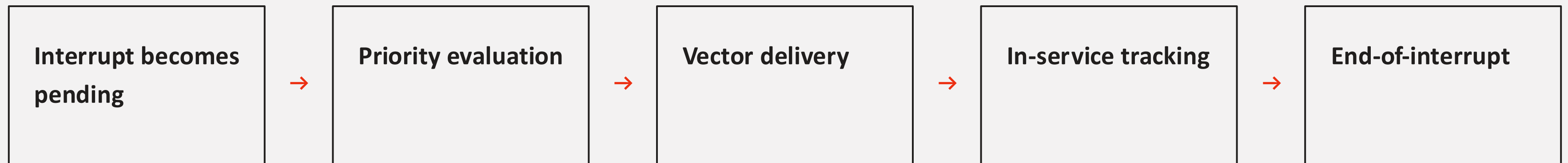
Ordinarily an EPT violation exits to the VMM. With the virtualization-exception facility, selected EPT violations instead become guest exception #VE on vector 20.

Hardware writes the information that would have accompanied the VM exit into a virtualization-exception information area, then delivers #VE to the guest if the required conditions hold. An enlightened guest component can resolve the condition locally without a round trip.

The VMM retains control over convertibility through EPT metadata and VMCS controls. Misconfigurations still exit, and violations from asynchronous facilities such as Intel PT or PEBS are not converted.

APIC virtualization removes exits from interrupt bookkeeping

Without acceleration, guest accesses to its APIC frequently require exits so the VMM can maintain a virtual interrupt controller. APIC virtualization lets hardware emulate many APIC accesses, track virtual interrupt state, and deliver virtual interrupts without returning to root operation.



If each arrow exits, the cost can dominate I/O-heavy workloads. Chapter 28 introduces the facility; Chapter 32 supplies the detailed APICv and posted-interrupt machinery.

Asynchronous producers need address virtualization too

Not every memory access is issued directly by a guest instruction. Facilities such as Intel Processor Trace and Trace and architectural PEBS can write trace or sampling records asynchronously.

VMX controls can make their output addresses guest-physical addresses translated through EPT. This prevents a guest from configuring a tracing facility to write directly into arbitrary machine memory.

TRACE-ADDRESS PRE-TRANSLATION

Intel PT adds a buffering problem: an EPT violation discovered only when buffered trace data is finally written could lose data during the resulting exit. TAPT validates and caches the translation before the buffer must be drained, so any necessary exit occurs early enough for software to repair the mapping without corrupting the trace stream.

PASID translation virtualizes accelerator identities

ENQCMD and ENQCMDs can submit work tagged with a process address-space identifier. A guest PASID must not automatically become the same numeric PASID in the host, because the host namespace is shared among VMs and devices.

guest PASID → PASID translation hierarchy → host PASID

Missing, invalid, or reserved entries cause a VM exit before the enqueue store completes. This is the same design pattern as EPT at a different layer: let the guest manage a realistic private namespace, then translate it into a host-owned global namespace at the hardware boundary.

Guest timer: virtual deadlines without host timer emulation

With APIC-timer virtualization, a guest write to `IA32_TSC_DEADLINE` can update a VMX guest deadline rather than the physical APIC timer. The written deadline is interpreted using the active TSC offset and scaling configuration.

- 1 Marks the virtual timer vector pending in the virtual APIC
- 2 Updates virtual-interrupt priority state
- 3 Evaluates whether the interrupt can be delivered
- 4 Clears the one-shot guest deadline

The guest sees a timer defined relative to its virtual TSC. Hardware already knows the time transformation and virtual-APIC state, so it can connect the two without exits on every deadline write and expiration.

VMX-PREEMPTION TIMER	GUEST TIMER
Owned by the hypervisor	Owned by the guest-visible virtual APIC
Regain control for scheduling or policy	Deliver the guest's timer interrupt

Unrestricted guest: boot states without contortions

Early VMX required `CR0.PE=1` and `CR0.PG=1` in VMX operation, which made real-address mode and unpaged protected mode difficult to represent directly.

With the unrestricted-guest control, non-root execution may use `CR0.PE=0` or `CR0.PG=0`. Firmware, boot loaders, and early operating-system code can run in their natural modes.

```
CR0.PG = 0
```

```
guest linear = guest-physical
```

```
guest-physical → EPT → machine physical
```

EPT must be enabled with unrestricted guest. Relaxing the guest's mode constraints is safe only because a separate translation layer still protects host memory. The guest can behave like a reset-era x86 machine without acquiring raw machine-physical access.

06

VM entry is a validated transaction

Entry is a transaction with staged validation

VMLAUNCH and VMRESUME do much more than load and change a mode bit. VM entry has to prove that the processor can safely install the requested guest context and that a future VM exit has a valid host destination.

This order reflects a safety hierarchy. Before touching guest state, the processor verifies it has a valid way back to the host. Once guest loading has begun, failure can no longer be reported as though the instruction simply did nothing.

- | | |
|---|--|
| 1 | Basic checks on the instruction's execution context and the current VMCS |
| 2 | Validate execution, exit, and entry controls, and host state |
| 3 | Check guest state while loading it, clearing address-range monitoring |
| 4 | Process the VM-entry MSR-load area |
| 5 | For VMLAUNCH , change the VMCS launch state to launched |
| 6 | Perform trace-address pre-translation when required |
| 7 | Inject a configured event or establish a pending MTF exit |

VMLAUNCH **versus** VMRESUME

VMLAUNCH requires a clear VMCS and is used for its first successful entry after VMCLEAR.

VMRESUME requires a launched VMCS and is used for entries.

Using the wrong instruction fails rather than guessing the caller's intent.

First entry and re-entry are different lifecycle events. Separating them lets hardware maintain implementation-private launched state and optimize the common resume path. VMCLEAR explicitly returns the object to the pre-launch state and flushes processor-owned VMCS information to memory.

ENTRY IS ALSO REJECTED UNDER HOST MOV-SS BLOCKING

That blocking describes a transient one-instruction stack-update window in the current host execution stream. It is not guest state and should not leak across the transition. Failing locally keeps the boundary unambiguous rather than carrying a half-completed host stack-update condition into non-root operation.

Four ways entry can fail

FAILURE CLASS	EXAMPLE	WHERE CONTROL GOES	REPORT MECHANISM
Ordinary architectural fault	VMLAUNCH at the wrong privilege level	Normal host fault handler	#UD or #GP
VMfailInvalid	No current VMCS, or current VMCS is a shadow VMCS	Instruction following VM entry	RFLAGS.CF=1
VMfailValid	Wrong launch state, invalid controls, bad host state	Instruction following VM entry	RFLAGS.ZF=1 plus VM-instruction error
Late VM-entry failure	Invalid guest state or failure loading an entry-list MSR	Host entry point from the VMCS	Exit-reason bit 31 plus reason 33, 34, or 41

On successful VM entry, control does not fall through to the next host instruction at all.

Why CF and ZF encode different failures

CF = 1

The processor could not identify a usable ordinary current VMCS from which to obtain an error field.

ZF = 1

There was a current ordinary VMCS, so the processor can record a record a VM-instruction error in it.

`VMfailInvalid`: there is no valid VMCS context for detailed reporting

`VMfailValid`: a VMCS exists, but its lifecycle or contents are invalid

The names describe whether the VMX instruction had a valid VMCS reporting context, not whether the entire virtual machine is valid.

Why late failures look like exits

Guest checks and guest-state loading may occur in parallel or in any order. A bad segment field or MSR-list entry may be discovered only after part of the requested guest context has reached the processor.

Continuing at the instruction after VMLAUNCH with the old old host register environment would be unsafe; that environment may no longer be intact. The processor loads the configured host-state area as it would on a and reports a VM-entry-failure reason.

Transactional recovery without pretending full rollback is possible. Once the transition has partially committed, recovery uses the already-validated host landing context.

EXIT-LIKE, BUT NOT AN ORDINARY EXIT

Guest state is not saved back to the guest-state area

The VM-exit MSR-store list is not processed

Most exit-information fields are not updated

The event-injection valid bit is not cleared

Host state and the VM-exit MSR-load list are used to recover

Why host state is validated before guest state

The host-state area describes the recovery environment for every future exit and for late entry failures. The processor checks it before beginning guest loading.

- Valid fixed-bit settings in host CR0 and CR4
- Legal and appropriately aligned CR3 state
- Canonical host RIP, RSP, and relevant bases on Intel 64 processors
- Segment selectors compatible with the normalized host segmentation model
- A valid TR selector and base
- Coherent address-space-size, paging, PAE, PCID, FRED, and EFER relationships
- Valid values in any control-gated host fields that will be loaded on exit

Do not start a guest unless the processor already knows it can construct a usable hypervisor when the guest stops.

This makes the exit path dependable even when the guest stops guest stops because of an unexpected fault or asynchronous asynchronous event.

Control validation is dependency validation

VMX controls are not independent feature toggles. Many controls imply other controls, address fields, or capabilities.

Unrestricted guest requires EPT

Virtual NMIs requires NMI exiting

Secondary controls are meaningful only when the primary activation bit is enabled

Tertiary controls likewise require their activating primary control

Bitmap and table addresses must be aligned and within the supported physical-address width

Counts for MSR areas must be consistent with address arithmetic and allowed sizes

An enabled facility makes its associated control-gated fields subject to validation

WHY THIS MAKES SENSE

A collection of individually legal settings can still describe an impossible machine. Entry validation checks the configuration as a graph of dependencies so hardware never begins execution with half of a feature enabled.

Guest state must describe one coherent architectural instant

CR0.PG=1 requires CR0.PE=1	Segment types, DPLs, present bits, unusable bits, and CS/SS relationships must describe a legal context
An IA-32e guest requires paging and PAE state consistent with long mode	Reserved RFLAGS bits and mode-dependent bits must have legal values
CR4.PCIDE is not valid for a 32e guest	Interruptibility bits cannot describe contradictory short-lived states
RIP, SYSENTER pointers, FS/GS bases, and CET pointers must satisfy canonical-address rules	Activity state must be supported and compatible with event injection
PAE PDPTEs must be valid when the selected paging mode requires them	

DESIGN READING

Software restore code normally reaches unusual states through a sequence of architecturally checked instructions. VM entry installs many components at once, bypassing that normal sequence. The processor therefore has to validate the final cross-register invariants explicitly.

A VMCS is not simply a bag of values that hardware blindly copies.

Loading guest state is selective, not a byte-for-byte copy

VM entry loads guest control registers, segment caches, descriptor tables, RIP, RSP, RFLAGS , non-register state, and control-gated MSRs. Some bits are fixed, ignored, derived from controls, or normalized.

The VMCS describes architectural intent. Entry logic maps that intent to the processor's valid internal state.

Some reserved or architecturally fixed CR0 and DR7 bits are not loaded directly

EFER long-mode state is tied to the IA-32e guest control and paging state

FS.base and GS.base correspond to both segment-cache state and architectural MSRs

The segment unusable bit controls whether other cached attributes matter

PDPTEs may be loaded from VMCS fields or through the paging structures

Activity and interruptibility fields recreate hidden boundary state

The VM-entry MSR-load list handles the long tail

After ordinary guest-state loading, VM entry processes a list of 16-byte MSR entries:

bits 31:0	MSR index
bits 63:32	reserved, must be zero
bits 127:64	value to load

Entries are processed in order. Invalid indexes, reserved bits, disallowed special MSRs, or a value that would fault cause a late VM-entry failure.

Ordering matters. A dedicated guest-state MSR field is loaded first; if the same MSR is legally present in the entry MSR-load list, the list value can overwrite it.

Dedicated fields cover frequently used, mode-coupled state that benefits from explicit validation. A count-and-address list supports additional MSRs without expanding the fixed VMCS architecture every time a processor adds one.

The list should remain short. Every entry adds transition work, so a VMM normally includes only state that truly differs between host and guest and is safe for this mechanism.

Event injection happens after the guest context exists

If the injected-event field is valid, VM entry delivers the selected selected interrupt, NMI, exception, software event, SYSCALL, SYSCALL, or SYSENTER after guest state and applicable have been loaded and VM-execution controls are active. ordering ensures delivery uses the guest's environment:

Guest IDTR and IDT, or guest FRED configuration	Guest page tables and EPT
Guest segment and privilege state	Guest interruptibility state
Guest stack or shadow stack	Guest event-delivery controls

For hardware exceptions, NMIs, and external interrupts, the saved return address is the current guest RIP . For injected software events, the VM-entry instruction-length field supplies the increment needed to construct the architectural return point.

If the VMM first resumed the guest and then tried to interrupt it, at least one unrelated instruction might run. Atomic injection guarantees the event is the guest's first architectural action after the transition.

Reinjection does not re-trigger its own interception

Suppose the exception bitmap intercepts `#PF` and the VMM decides to reinject a page fault. The injected fault does not cause a VM exit merely because the bitmap bit is set. Otherwise reinjection would loop forever.

configured injection

- original event delivered in guest
- interception does not reject it
- a new nested condition may exit

Delivery can still encounter a new problem: pushing the frame might cause an EPT violation, or the guest IDT descriptor might generate a nested exception selected by the bitmap.

ORIGINAL-EVENT INFORMATION

VMM injects guest interrupt vector 50

- processor begins pushing frame
- stack write causes EPT violation
- VM exit reports EPT violation
- original-event fields keep vector 50

The VMM can repair the EPT mapping and arrange to complete delivery without inventing or duplicating the interrupt. This preserves both VMM authority and forward progress.

Entry can succeed and still exit before one guest instruction

A successful VM entry does not promise that an ordinary guest guest instruction will execute. Conditions evaluated immediately immediately afterward may cause an exit or event delivery:

A preemption timer loaded with zero	A pending MTF exit
An already-open interrupt window window	A pending debug exception
An open NMI window	A virtual interrupt ready for delivery
A TPR-threshold condition	Feature-specific preparation such as TAPT encountering a translation problem

This is not a failed entry. The processor constructed the guest context, reached the relevant guest boundary, and honored the highest-priority pending condition.

"No guest instruction retired" does not imply VMLAUNCH or VMRESUME failed. The exit reason and VM-entry-failure bit distinguish a real entry failure from an immediate post-entry exit.

ACTIVITY AND INTERRUPTIBILITY

Without injection, the activity-state field determines whether the processor becomes active, halted, shutdown, or wait-for-SIPI. With injection, it becomes active because it must deliver the event. Entry restores a point on an execution timeline, not merely register contents.

07

VM exit is transfer plus explanation

The five steps of a VM exit

- 1 Record exit information
- 2 Save VMCS-managed guest state
- 3 Process the VM-exit MSR-store area
- 4 Load or synthesize host state, clearing address-range monitoring
- 5 Process the VM-exit MSR-load area

Execution then begins at `HOST_RIP` on `HOST_RSP`.

Guest general-purpose registers are still in the physical registers until the software trampoline saves them.

Architectural state is defined before the save begins

Chapter 30 first defines what state exists at the instant an exit commences, because different causes stop execution at different phases.

FAULT-LIKE INSTRUCTION EXIT

The instruction did not execute. Saved RIP points at it; architectural side effects are absent.

TRAP-LIKE INSTRUCTION EXIT

The operation completed. Saved state includes its effects; saved RIP usually points at the following instruction.

DIRECT EVENT EXIT

The selected event exits before guest delivery. Normal delivery side effects generally do not occur; a directly exiting page fault does not update CR2.

INDIRECT EVENT EXIT

Delivery began but hit another exit condition. Incomplete delivery may have written part of a guest stack frame. The processor reports both causal layers.

Precise virtualization requires the VMM to continue from the same architectural boundary bare hardware would have reached. "Approximately before the event" is not sufficient for nested faults, debugging, NMI blocking, or restartable instructions.

Exit information is a structured handoff

INFORMATION
Exit reason
Exit qualification
Guest linear address
Guest-physical address
Exiting-event identification and error code
Original-event identification and error code
Instruction length
Instruction information
VM-instruction error

Not every field is valid for every reason. A handler must dispatch on the exit reason and use only the fields defined for that path. An ordinary VM

QUESTION ANSWERED
What broad condition returned control?
What reason-specific details apply?
Which guest virtual address participated?
Which intermediate physical address participated?
Which event directly caused the exit?
Which event was being delivered when another condition exited?
How long was the intercepted instruction or software event?
Which registers, addressing mode, segment, and operand form were decoded?
Why did a VMX instruction fail without an exit?

Design reading. Variable-format qualification avoids inflating the VMCS with a separate record for every exit class. Hardware reports a common

Why instruction information saves more than an opcode

For selected exits, hardware reports decoded addressing facts:

Address size and scaling

Segment register

Base and index registers

Whether operands are registers or memory

Operand registers for VMX or descriptor-table instructions

The processor has already decoded this to execute the instruction. Reporting it avoids a second decoder in the VMM and avoids fetching instruction bytes from memory that may itself be inaccessible or changing.

$$\begin{aligned} \text{guest RIP} &= \text{guest RIP} \\ &\quad + \text{VM-exit instruction length} \end{aligned}$$

That advance is appropriate only when the handler has completed the instruction's guest-visible effect. Retrying the instruction instead leaves RIP unchanged.

Guest RIP is chosen according to the exit boundary

The value saved in the guest RIP field is not governed by one universal "current instruction" rule.

An intercepted fault-like instruction saves the address of that instruction

An external interrupt or hardware exception saves the return pointer that normal delivery would have used

Interrupt-window and NMI-window exits save the boundary where execution would otherwise continue

A trap-like exit saves state after the triggering operation

Exits from HLT or MWAIT inactive states normally preserve the post-instruction continuation point

Software-exception exits retain the instruction boundary needed for correct reinjection semantics



The saved RIP is a restart contract. It tells the VMM whether resumption should retry an operation, continue past it, or deliver an event at the captured boundary.

VM exit saves VMCS-managed state, not the whole vCPU

SAVED BY HARDWARE

CR0, CR3, and CR4	Debug state when configured
Guest segment caches and descriptor tables	Activity and interruptibility state
RIP, RSP, and RFLAGS	Pending debug state under defined conditions
SYSENTER state and control-gated MSRs	Remaining preemption-timer value when requested
Cached PDPTEs when applicable	Virtual-interrupt status and guest deadline when supported

It does not save ordinary general-purpose registers into the VMCS, and it does not automatically switch x87, SSE, AVX, or other XSAVE state.

THE COMPLETE vCPU IMAGE

VMCS-managed state

+ software-saved GPRs

+ software-managed extended state

+ virtual-device and scheduler state

GPRs remain live in the processor until the host entry trampoline stores them in the software vCPU structure. The VMM and host OS manage the larger extended state according to their own scheduling and ownership policy.

The exit MSR lists form a context-switch pair

After saving ordinary guest state, VM exit can read selected MSRs into the VM-exit MSR-store area. After loading the host-state area, it can load selected host MSRs from the VM-exit MSR-load area. This supports MSRs such as a guest's IA32_LSTAR that have no dedicated VMCS field.

```
guest runs with guest LSTAR
→ exit MSR-store saves guest LSTAR
→ host-state load establishes basic
  host context
→ exit MSR-load installs host LSTAR
```

The matching VM-entry list can restore guest LSTAR next time. The lists are powerful but dangerous: an invalid store or load during VM exit causes a VMX abort, because the processor is already committed to leaving the guest and cannot safely report an ordinary guest-visible fault. Entry has a validated host recovery target, so some entry failures are recoverable. Exit is itself the recovery path. Failure of that path is catastrophic.

Host state is a landing environment, not the old snapshot

LOADED FROM THE HOST-STATE AREA

Host CR0, CR3, CR4	GDTR and IDTR bases
HOST_RSP and HOST_RIP	TR state
Required segment selectors and bases	SYSENTER state, enabled optional MSRs and security state

SYNTHESIZED, NOT RESTORED

```
RFLAGS          = 0000000000000000002H
DR7              = 0000000000000000400H
IA32_DEBUGCTL    = 0
```

The host begins active, without STI or MOV SS blocking and without pending debug exceptions. Host segmentation is segmentation is normalized rather than restored from a complete guest-style descriptor cache.

This is why the host-state area is smaller than the guest-state area. The guest can be paused in many awkward architectural states. The VMM chooses one conventional entry point and promises that it satisfies stricter rules.

Why interrupts begin disabled in the host

Because synthesized host RFLAGS has $IF=0$, an ordinary maskable interrupt cannot preempt the exit trampoline before it has saved guest registers and established its software frame.

$DF=0$ ensures host string operations use the conventional forward direction, and $TF=0$ prevents guest or pre-entry single-step state from debugging the host handler.

The host's old flags are not automatically restored because because VM exit is not a return to an interrupted host instruction. The VMM's run wrapper reconstructs any higher-level software continuation it needs.

```
host_rip:
    push rax
    push rbx
    push rcx
    ; capture remaining guest registers
    ; establish host ABI state
    ; optionally enable interrupts later
```

The exit handler is a reason-driven state machine

```
save guest GPRs
read basic exit reason
switch reason:
    CPUID:
        synthesize registers
        advance guest RIP

    intercepted MSR:
        validate/emulate
        advance RIP or inject #GP

    EPT violation:
        inspect qualification and guest-physical address
        repair mapping, emulate access, inject an event, or stop VM

    acknowledged external interrupt:
        read physical vector from exiting-event information
        dispatch host IRQ, service source, and issue EOI
        queue a separate virtual interrupt if appropriate

    unacknowledged external interrupt:
        preserve guest frame, then allow normal host IDT delivery

    physical NMI:
        dispatch host NMI machinery with correct unblocking semantics
```

The hard part is not the switch statement. It is honoring the restart semantics reported by each exit: which state already changed, which event remains pending, whether RIP advances, and whether resumption

VMX abort: the exit path could not construct a valid host

A VMX abort is not an ordinary VM exit and not a guest exception. The processor encountered a problem from which the architectural VM-exit mechanism could not safely recover.

Failure while saving guest MSRs from the exit-store list
Failure loading host MSRs
Invalid host PAE PDPTes
Corruption that prevents the current VMCS from supporting exit
An incompatible host address-space configuration
Certain machine-check events during exit

The processor writes a nonzero abort indicator at byte offset 4 of the responsible VMCS region. Outside SMX operation, it then enters a special shutdown state from which only reset recovers.

An ordinary exception needs a valid execution context in which to run its handler. A VMX abort occurs because the processor cannot reliably finish constructing that context. Continuing with a mixture of guest and host state would be worse than stopping.

The abort indicator is memory-resident so another logical processor or post-reset diagnostic code can inspect why the failed CPU stopped.

08

Round trips and design patterns

Round trip 1: an intercepted CUID

1. Guest executes CUID.
2. Ch. 28: CUID causes a fault-like VM exit.
3. Ch. 30 saves guest RIP pointing at CUID.
4. Exit reason identifies CUID; instruction length identifies its size.
5. Host trampoline saves guest GPRs.
6. VMM computes virtual CUID values into saved RAX/RBX/RCX/RDX.
7. VMM advances guest RIP by the exit instruction length.
8. VMM executes VMRESUME.
9. Ch. 29 validates and reloads the guest.
10. Guest continues after CUID with the virtual results.



The actual instruction never executes with raw host semantics.

Round trip 2: a pending interrupt while $IF=0$

1. Host queues a virtual interrupt for the vCPU.
2. Guest RFLAGS.IF is zero, so injection would be illegal.
3. VMM enables interrupt-window exiting and resumes.
4. Guest runs until $IF=1$ and STI/MOV-SS blocking is absent.
5. Ch. 28 causes an interrupt-window VM exit.
6. VMM disables window exiting and configures VM-entry injection.
7. Ch. 29 loads guest state and injects the interrupt atomically.
8. Guest begins in its IDT or FRED handler.

The VMM does not poll after every instruction, and the interrupt does not arrive at an illegal boundary.

Round trip 3: an exit during injected-event delivery

1. VMM configures injection of an external interrupt.
2. VM entry loads the guest and begins event delivery.
3. The processor tries to push the guest interrupt frame.
4. The stack access encounters an EPT violation.
5. VM exit reports the EPT violation as the immediate cause.
6. Original-event fields preserve the injected interrupt.
7. VMM repairs the EPT mapping.
8. VMM arranges retry without losing or duplicating the interrupt.

This is why Chapters 29 and 30 devote so much space to events "incident to" other events.

Causal history is part of resumable machine state.

The patterns behind Chapters 28 through 30

Native execution is the default

Performance comes from choosing the smallest interception set consistent with isolation and emulation requirements.

Entry validates a future exit before trusting the guest

The host landing state is the recovery mechanism for both normal exits and late entry failures.

Transition work is capability- and control-gated

Optional state is loaded, saved, or validated only when the facility is supported and enabled. Old hypervisors stay compatible and nobody pays for unused features.

Hardware handles predicates; software handles policy

Bitmaps, masks, exact-value targets, priority fields, and time thresholds answer fast-path questions. Device models, scheduling, and security policy exit to software.

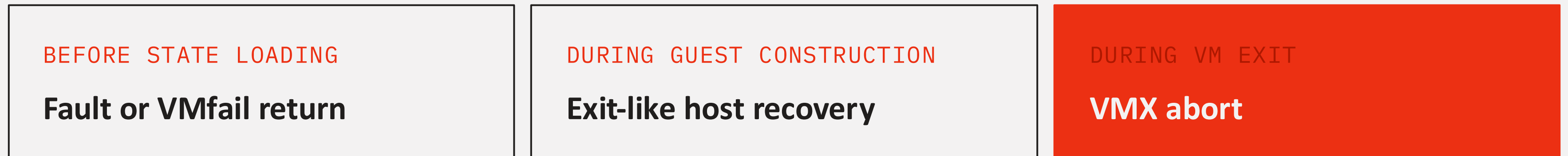
Hidden boundary state is first-class state

Interrupt shadows, activity state, pending debug events, virtual-NMI blocking, and partial event delivery determine what can happen next even when visible registers match.

The host side is intentionally normalized

The guest needs fidelity; the host needs a safe landing pad. Synthesizing host flags and segment properties makes the first host instruction predictable.

Failure severity depends on the point of commitment



The later the failure occurs, the less old context remains available for recovery.

A practical reading checklist

1	Is the operation native, modified, faulting, fault-like exiting, or trap-like exiting?	6	Which exit-information fields are defined for this reason?
2	Which control, bitmap, mask, or capability selected that behavior?	7	Was another event already being delivered?
3	At what architectural boundary does the transition occur?	8	Does the handler need to advance RIP , retry, inject an event, or block the vCPU?
4	Has the triggering instruction or event already changed guest state?	9	Which state is in the VMCS, which remains in registers, and which belongs to software?
5	What does saved guest RIP mean for retry versus continuation?	10	If entry fails, is it a fault, VMfailInvalid, VMfailValid, or a late exit-like failure?

Those questions turn the manual's many special cases into one consistent task: identify the exact boundary and preserve its semantics.

VMX makes virtualization fast by allowing guest code to use the ordinary execution engine, and safe by evaluating compact hardware policy at the points where guest behavior could affect the machine or require emulation. VM entry is a staged, validated installation of a guest context with a prevalidated host recovery target. VM exit captures the guest at a precisely defined boundary, records enough causal information for software to act correctly, and constructs a normalized host landing environment.

Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3

System Programming Guide, Order Number 325384-092US, June 2026.

PRIMARY READING

Chapter 28 — VMX non-root operation, exit causes, modified instruction behavior, non-root features, unrestricted guests

Chapter 29 — VM-entry sequencing, validation, state loading, MSR loading, event injection, entry failures

Chapter 30 — exit boundaries, exit-information recording, guest-state saving, host-state loading, MSR switching, VMX aborts

SUPPORTING MATERIAL

Chapters 26–27 — VMX lifecycle, VMCS organization, state fields, controls

Chapter 31 — detailed EPT, VPID, address-translation behavior

Chapter 32 — APIC virtualization, virtual interrupts, posted interrupts

Appendix A — VMX capability reporting

Appendix C — basic VM-exit reasons

Rationale marked as a **design reading** is engineering interpretation of the specified behavior, not a claim that Intel documented the historical or microarchitectural motive. Production hypervisor code must follow the manual's exact pseudocode, priority rules, capability checks, reserved-bit requirements, errata, and processor-specific documentation.