

Lecture 03 Intel VMX

A design-oriented guide to Chapters 26 and 27 of the Intel 64 and IA-32

Architectures Software Developer's Manual, Volume 3C

Order Number 325384-092US, June 2026

Contents

01 Why VMX adds another execution axis

02 The VMX lifecycle and enablement

03 The VMCS as a bidirectional contract

04 The guest-state area

05 The host-state area

06 Controls, transitions, and exit information

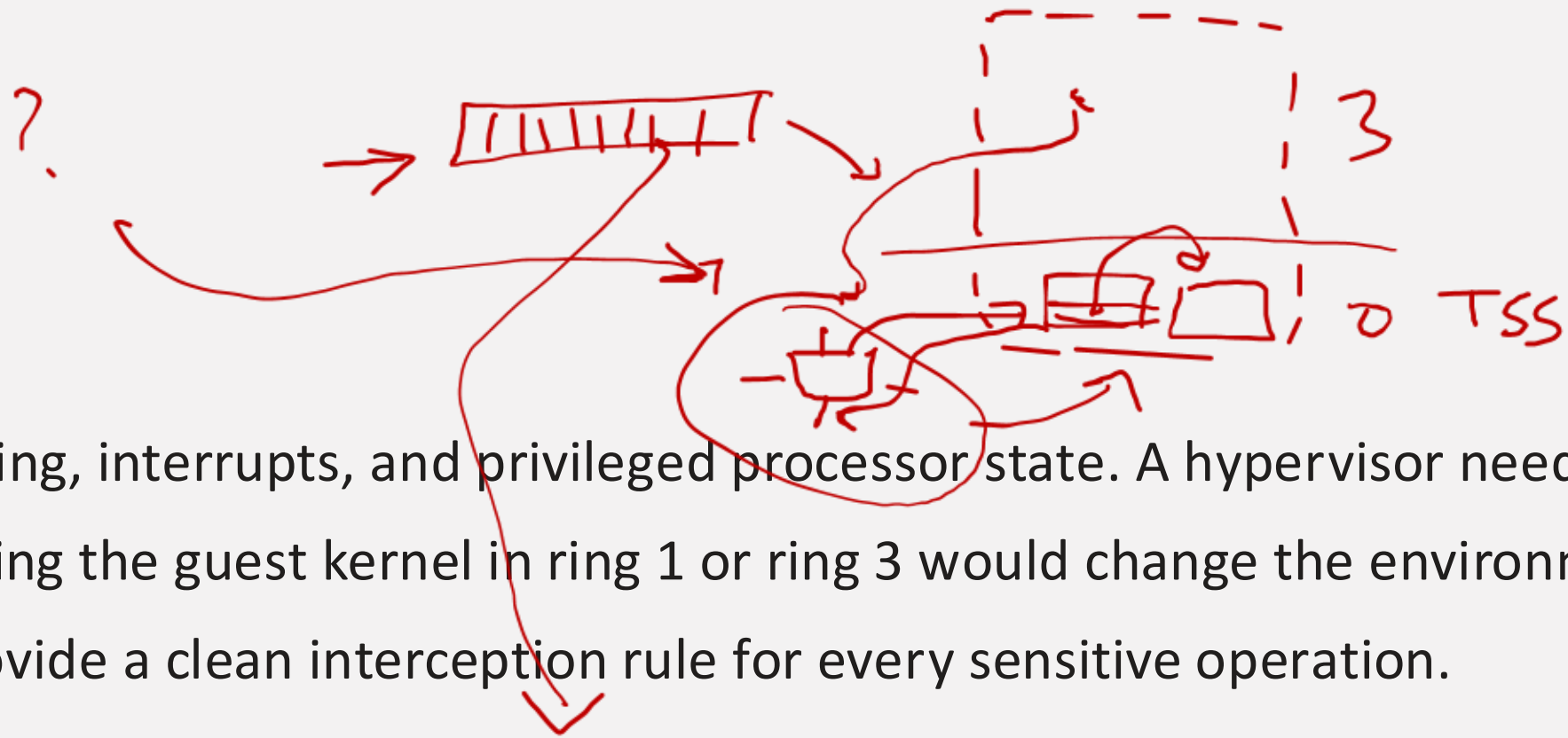
01

VMX lets ordinary guest code run directly, while giving the hypervisor a hardware-enforced contract for state, interception, and transitions. The VMCS is that contract for one virtual CPU.

Intel's manual is precise about *what* VMX does, but a first reading can feel like walking through an inventory. The inventory becomes easier to remember when each item is tied to the problem it solves.

Rings were not enough

OS kernels expect CPL 0, where they manage paging, interrupts, and privileged processor state. A hypervisor needs ultimate control over those same resources. Putting the guest kernel in ring 1 or ring 3 would change the environment the kernel was written for, and would still not provide a clean interception rule for every sensitive operation.

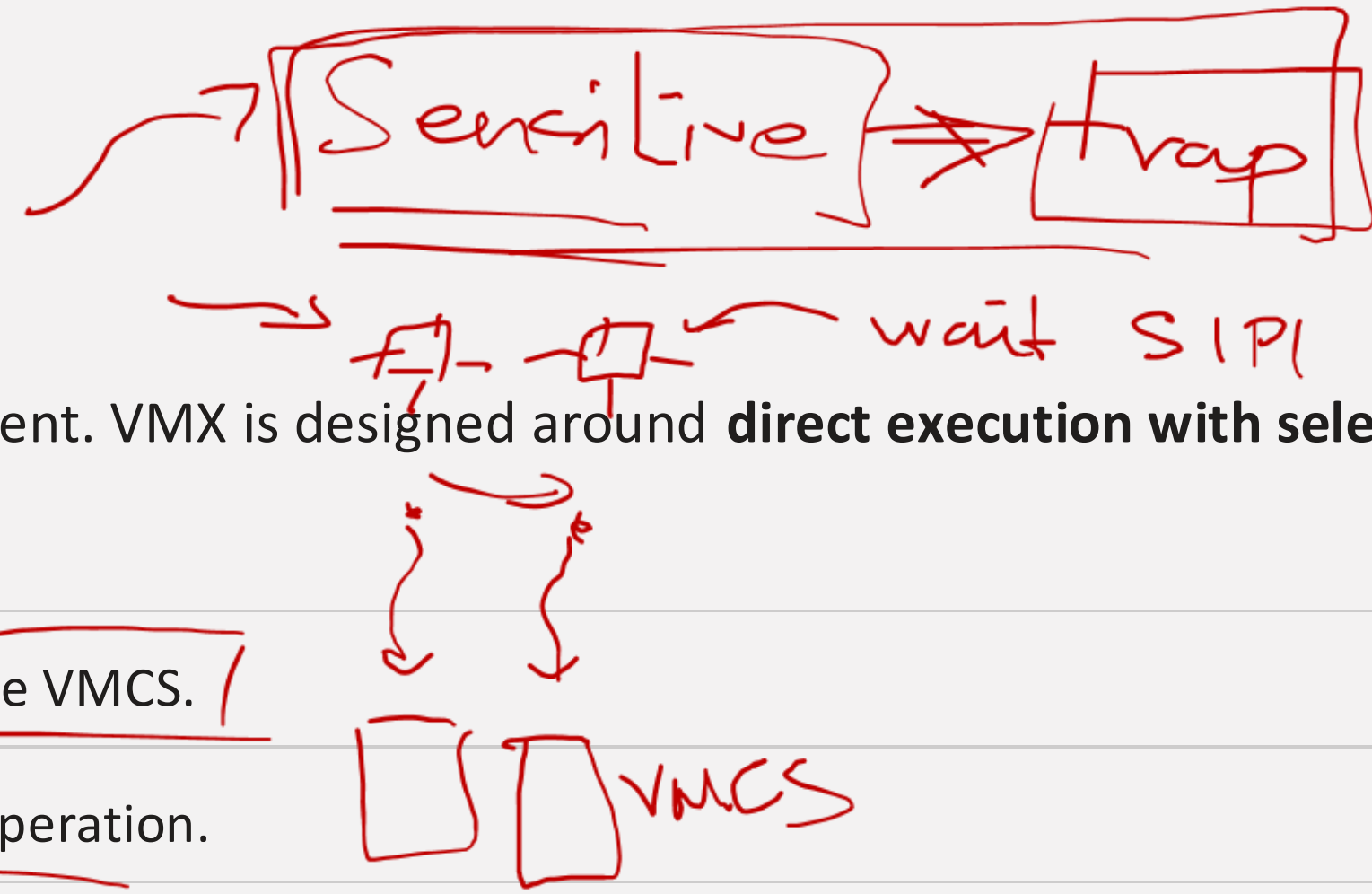


VMX OPERATION	TYPICAL OCCUPANT	CPL CAN STILL BE
→ Root ←	Hypervisor / VMM	0 through 3
→ Non-root	Guest OS and its applications	0 through 3

A guest kernel therefore executes in VMX non-root operation at CPL 0. Its ring-based assumptions remain largely intact, but VMX controls can still restrict or redirect its behavior.

Why not emulate every instruction?

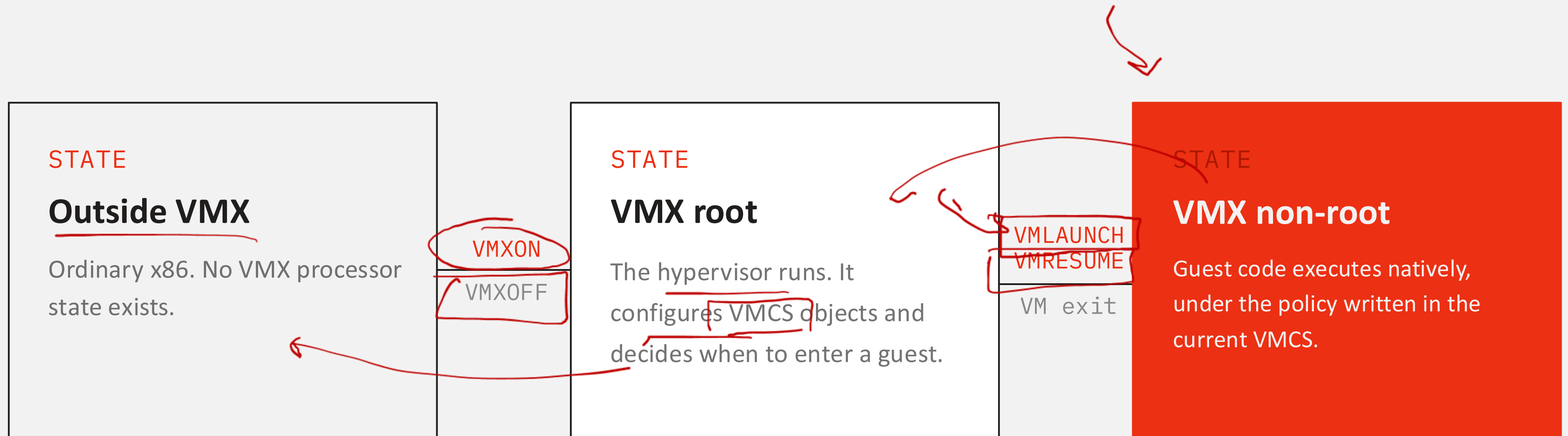
Because most instructions do not need hypervisor involvement. VMX is designed around **direct execution with selective interception**.



- 1 The hypervisor establishes guest state and a policy in the VMCS.
- 2 A VM entry moves the logical processor into non-root operation.
- 3 Guest instructions execute natively until a configured instruction, event, or condition causes a VM exit.
- 4 A VM exit records why it happened, saves VMCS-managed guest state, loads host state, and transfers control to the hypervisor.
- 5 The hypervisor handles the event and enters the guest again.

■ **Design reading.** The boundary is event-driven because transition cost matters. The controls exist to make the fast path as wide as the hypervisor can safely allow.

Three states, four transitions



The processor does not return from VMLAUNCH to the following host instruction. The next root-mode instruction begins at `HOST_RIP` after a VM exit.

Why is there no simple "I am a guest" bit?

Section 26.3 notes that no software-visible bit directly indicates VMX non-root operation. This keeps the mechanism separate from the guest's ordinary architectural architectural state.

A hypervisor may deliberately identify itself through CPUID, CPUID, paravirtual interfaces, or emulated devices. VMX does not force detection through a mode flag.

virt 10

DESIGN READING

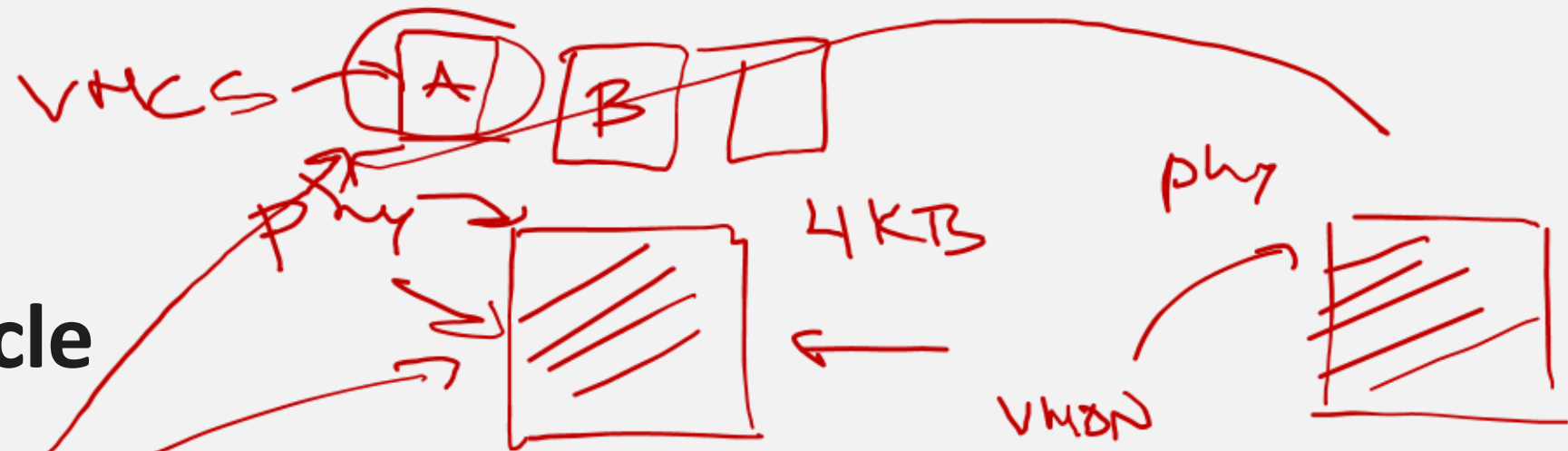
Transparency is useful for running unmodified software. More importantly, a guest-mode bit would itself become another value that had to be virtualized consistently across instructions, saved state, and nested virtualization.

02

VT-x

The VMX lifecycle and enablement

VMX has an explicit lifecycle



discover → enable → VMXON → configure VMCS → enter/exit guests → VMCLEAR → VMXOFF

DISCOVER

CPUID answers the binary question;
capability MSRs describe the details.

CONFIGURE

One VMCS per virtual CPU, selected with
with VMPTRLD.

ENABLE

IA32_FEATURE_CONTROL carries a
platform policy that firmware can
lock.

RUN

VMLAUNCH for the first entry,
VMRESUME thereafter.

VMXON

CR4.VMXE grants permission;
VMXON creates live VMX processor state.

RELEASE

VMCLEAR hands a VMCS back to memory; VMXOFF ends the CPU's VMX
VMX lifetime.

CPUID plus capability MSR: discover, do not assume

CPUID.01H:ECX.VMX[5] ←

Reports basic VMX support. A small binary question, answered before software touches VMX at all.

VMX capability MSRs ←

Describe control-bit legality, VMCS size and revision, supported activity states, fixed CR0/CR4 bits, and optional features.

VMFUNC

WHY THIS SPLIT MAKES SENSE

CPUID answers the small binary question needed before touching VMX. MSRs provide an extensible description of a much larger feature set. A hypervisor can run on multiple processor generations by constructing a supported policy at boot instead of compiling in one model's control words.

IA32_FEATURE_CONTROL: why firmware gets a lock

VMXON is gated by this MSR. Its lock bit prevents further writes until a power-up reset; separate enable bits govern VMXON inside and outside SMX operation.

Enabling a hypervisor changes the machine's security boundary. Firmware can establish a platform-wide policy before the operating system starts, then lock it so later software cannot silently rewrite that decision.

The cost is operational: if firmware locks VMX off, the OS cannot repair the setting without rebooting through firmware that enables it.

```
power-up reset
→ firmware chooses whether
  VMX is permitted
→ firmware sets
  IA32_FEATURE_CONTROL.Lock
→ the setting cannot change
  again until the next
  power-up reset
```

Locking the decision early makes the boot chain, rather than whichever ring-0 component runs last, the authority.

A concrete case: VMX only after a measured launch

A platform may want to permit virtualization only after an Intel TXT measured launch. The policy might be:

```
VMX inside SMX    = allowed
VMX outside SMX   = forbidden
Lock               = 1
```

SMX means Intel's Safer Mode Extensions — the mechanism used by Intel TXT to start a measured execution environment.

THE INTENDED INVARIANT

Only code launched through the measured TXT path can become the machine's hypervisor.

INTENDED SECURITY FLOW

- 1 Firmware configures IA32_FEATURE_CONTROL.
- 2 It permits VMXON only inside SMX.
- 3 It locks the MSR.
- 4 TXT measures an approved hypervisor into TPM state.
- 5 The measured hypervisor executes VMXON.
- 6 Local policy or a remote verifier can rely on the intended hypervisor having been launched.

The separate SMX and non-SMX enable bits exist because "virtualization is allowed" and "only a measured launch may establish the hypervisor" are meaningfully different security policies.

What could happen without the lock?

Suppose the ordinary operating-system kernel is later compromised. If `IA32_FEATURE_CONTROL` remained writable, the malicious kernel could do this:

```
enable VMX outside SMX
→ set CR4.VMXE
→ execute VMXON
→ install an unmeasured
  malicious hypervisor
→ place the existing OS
  underneath it as a guest
```

This is sometimes described conceptually as a "hyperjacking" or "Blue Pill" style attack: the running OS is silently moved into a VM controlled by malicious software.

RUNNING BELOW THE OS, IT COULD POTENTIALLY

- Intercept guest memory accesses.
- Observe or modify interrupts.
- Manipulate virtual devices.
- Hide memory from the compromised OS's own security tools.
- Present false register or hardware state to the OS.
- Survive attempts to inspect the system from inside that OS.

IF FIRMWARE HAS ALREADY LOCKED

```
VMX outside SMX = 0
Lock              = 1
```

The compromised kernel cannot change the policy. Its attempt to enable outside-SMX VMX is blocked until a power-up reset.

Simpler case: firmware disables VMX completely

A machine owner may configure firmware with:

```
VMX inside SMX    = 0
VMX outside SMX   = 0
Lock               = 1
```

REASONS COULD INCLUDE

- Reducing the available attack surface.
- Preventing unauthorized hypervisors.
- Meeting a deployment or compliance policy.
- Avoiding use of a vulnerable virtualization stack.
- Reserving virtualization for a different boot configuration.

Without the lock, privileged software could reverse that firmware decision after boot. With the lock, changing it requires returning through the firmware-controlled boot process.

WHY THIS MATTERS IF THE ATTACKER ALREADY HAS KERNEL PRIVILEGES PRIVILEGES

A kernel compromise is already serious, so this is defense in depth rather than a complete security boundary. But becoming the hypervisor gives the attacker a different and potentially more concealed level of control.

Once the original OS is a guest, security software inside it cannot necessarily trust what it sees. The malicious hypervisor controls the virtualized view presented to that software.

TWO DISTINCT POLICIES

"A known hypervisor measured by TXT may use VMX."

"Arbitrary ring-0 software must not start VMX."

What the lock does not do

The lock does not identify which software is authorized to execute VMXON.

```
VMX outside SMX = 1  
Lock             = 1
```

VMX is then permitted outside SMX for the entire boot. The lock prevents that decision from changing, but it does not distinguish a legitimate OS hypervisor from malicious ring-0 code.

IA32_FEATURE_CONTROL is best understood as a firmware-controlled, boot-long platform policy about *where* VMX may be activated — not a per-program permission system.

CR4.VMXE plus VMXON: why two steps?

Software first sets CR4.VMXE, then executes VMXON with the physical address of a VMXON region. VMXOFF leaves VMX operation; only then may software clear CR4.VMXE.

STEP ONE SEPARATES

Architectural permission

Permission to execute VMX instructions at all. A control-register bit, checked like any other.

FROM STEP TWO

Live VMX processor state

Creation of processor-owned VMX state, backed by the VMXON region and bounded by an explicit lifetime.

- That separation gives faults precise meanings and provides an explicit lifetime around processor-owned VMX state.



Why a VMXON region is per logical processor

Before VMXON , software allocates a naturally aligned region, writes the VMCS revision identifier into its first 31 bits, and supplies its physical address. The required size is implementation-specific but no larger than 4 KiB. Each logical processor must use a separate region, and software must not modify it between VMXON and VMXOFF.

The VMXON region is **not** a VMCS and does not describe a guest.

PER-CPU

Each CPU enters and leaves VMX independently, so the region's scope matches its lifetime.

PHYSICAL

A page-aligned physical region gives the processor a stable object that does not depend on the current virtual address space.

OPAQUE

An implementation can keep VMX bookkeeping there without exposing microarchitectural layout as an ABI.

Two objects, two lifetimes

OBJECT	SCOPE	SELECTED BY	PURPOSE
VMXON region	One logical processor while it is in VMX operation	<u>VMXON</u>	Establishes that processor's VMX lifetime and implementation-private bookkeeping
<u>VMCS region</u>	Normally one virtual CPU	<u>VMPTRLD</u>	Describes guest state, the host exit target, execution policy, and exit information

One VMXON region cannot replace the VMCS because a processor may run many vCPUs during one VMX lifetime. Conversely, a VMCS cannot replace the VMXON region because VMX must have processor-wide state before any particular guest object is selected.

Fixed CR0 and CR4 bits: protecting mode invariants

While VMX is active, some CR0 and CR4 bits may be required to be zero or one. Software discovers the valid masks through `IA32_VMX_CR0_FIXED0/1` and `IA32_VMX_CR4_FIXED0/1`. `VMXON`, VM entry, and VM exit refuse states that violate these rules.

```
(value & FIXED0) == FIXED0
(value & ~FIXED1) == 0
```

A bit that is 1 in `FIXED0` must be 1. A bit that is 0 in `FIXED1` must be 0. A bit permitted by both masks remains software-selectable.

WHY THIS MAKES SENSE

VM entry and exit are already complex state changes. Fixed bits let a processor generation define the base execution invariants under which its VMX machinery is valid.

Later processors added the *unrestricted guest* control, allowing a non-root guest to present states such as real-address mode even when root operation retains stricter requirements.

A recurring VMX pattern: keep a safe base contract, then add a capability-negotiated relaxation.

The fixed bits in practice

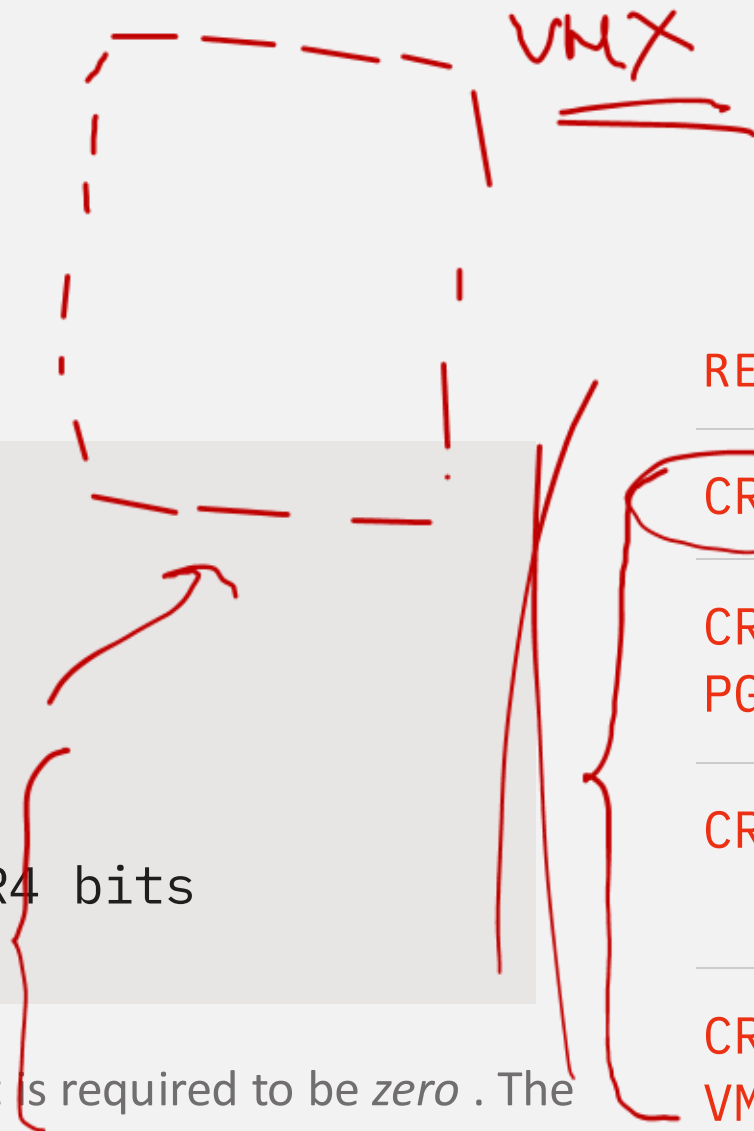
On typical Intel parts the masks read:

```
IA32_VMX_CR0_FIXED0 = 80000021H
IA32_VMX_CR0_FIXED1 = FFFFFFFFH

IA32_VMX_CR4_FIXED0 = 00002000H
IA32_VMX_CR4_FIXED1 = supported CR4 bits
```

CR0_FIXED1 being all ones means no CR0 bit is required to be zero. The constraint is only the three required-1 bits.

CR4_FIXED1 is the more active mask: it reports which CR4 features the processor supports at all, so any unimplemented bit is required to be zero.



REQUIRED TO BE ONE

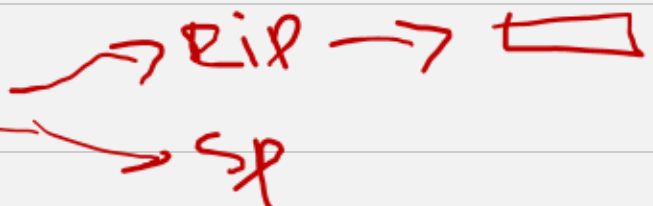
CR0 bit 0 · PE	Protection enable. Protected mode must be on.
CR0 bit 31 · PG	Paging. Paging must be on.
CR0 bit 5 · NE	Numeric error. FPU errors must report via #MF rather than the legacy external-interrupt path.
CR4 bit 13 · VMXE	Which is why CR4.VMXE cannot be cleared while in VMX operation — the fixed-bit rule enforces what the two-step ordering describes.

READ PE AND PG TOGETHER

They say the VMX transition machinery assumes a paged protected-mode environment — exactly the base execution invariant. The *unrestricted guest* control is what relaxes this for non-root operation, letting a guest present PE=0/PG=0 while root operation still obeys the stricter masks.

03

Six groups answer six different questions

VMCS GROUP	QUESTION ANSWERED
Guest-state area 	What architectural state should the guest resume with, and what should be saved on exit?
Host-state area 	Where and in what architectural environment should the exit handler begin?
<u>VM-execution controls</u>	<u>What may the guest do directly, and what should be intercepted or virtualized?</u>
<u>VM-exit controls</u>	What extra state transformation should happen on the way to the host?
<u>VM-entry controls</u> 	What extra state transformation or <u>event delivery</u> should happen on the way to the guest?
<u>VM-exit information</u> 	Why did control return, and what facts does the handler need?

State describes *what the two contexts are*. Controls describe *how and when to cross the boundary* . Exit information describes *what just happened*.

The organization is more than filing

VMCS for one vCPU

State

guest + host areas

What the two contexts are: the state the guest resumes with, and the environment the exit handler starts in.

Policy

execution, entry, exit

How and when to cross the boundary: what runs natively, what is intercepted, what is transformed in transit.

Diagnostics

exit reason + context

What just happened: the reason, the qualification, the addresses, and the event that was interrupted.

Three architectural words, then opacity

BYTE OFFSET	CONTENTS
0	Bits 30:0 — VMCS revision identifier · bit 31 — shadow-VMCS indicator
4	VMX-abort indicator
8 onward	Implementation-specific VMCS data

REVISION IDENTIFIER

Prevents the processor from consuming a VMCS image in an incompatible internal format; VMPTRLD fails on a mismatch. Intel can change the internal representation without changing the field interface.

SHADOW INDICATOR

The processor must know the object's role before interpreting the rest of it. Keeping the type beside the revision lets VMPTRLD reject an inappropriate object early.

VMX-ABORT INDICATOR

A VMX abort is a severe transition failure where ordinary exit reporting may be unavailable. A fixed, readable word is a last-resort diagnostic channel.

Why VMCS fields have encodings instead of offsets

Every VMCS field has a 32-bit architectural encoding.

14:13	Width: 16-bit, 64-bit, 32-bit, or natural width
11:10	Type: control, exit information, guest state, or host state
9:1	Index within that width/type class
0	Full access, or high-half access for a 64-bit field

WHY ENCODINGS MAKE SENSE

An offset would expose the private in-memory layout. A semantic encoding says what a field *is* while leaving the processor free to store it anywhere, cache it internally, or synthesize it.

Width and type bits also make decoding systematic, and the high-half high-half access form avoids requiring 64-bit operands from 32-bit 32-bit hypervisors.

Current, active, clear, and launched

CURRENT

The VMCS implicitly addressed by VMREAD, VMWRITE, VMLAUNCH, and VMRESUME on this logical processor. VMPTRLD selects it.

ACTIVE

Associated with a logical processor and potentially represented partly in processor-local state.

CLEAR

Not yet successfully launched since the last VMCLEAR.

LAUNCHED

Has completed a successful VMLAUNCH; later entries use VMRESUME.

```
VMPTRLD [addr-of-phys-addr]
      |
      +-> current VMCS pointer
```

```
VMWRITE ... updates fields in it
VMLAUNCH enters using it
VMRESUME re-enters using it
```

VMPTRLD means **load the VMCS pointer** . It does not enter the load guest registers, or copy the VMCS into architectural the control object that later VMX instructions will use.

The operand's extra indirection is worth noticing: the instruction reads a physical address *from memory* . The VMCS is identified physically because it must remain stable across changes to CR3 and virtual-address mappings.

One owner at a time

No VMCS may be active on more than one logical processor. To migrate one, the old processor executes `VMCLEAR` so processor-maintained data reaches memory and the VMCS becomes inactive; the new processor then uses `VMPTRLD`. Because `VMCLEAR` also resets launch state, the next entry is `VMLAUNCH`.

- The implicit current pointer keeps frequent VMCS operations compact.
- Exclusive active ownership lets hardware cache VMCS state without cross-CPU coherence rules for the opaque body.
- `VMCLEAR` is both a write-back/ownership handoff and a launch-state reset.
- Distinguishing `VMLAUNCH` from `VMRESUME` gives hardware a well-defined first-entry path; Intel notes that `VMRESUME` is generally expected to have lower latency.

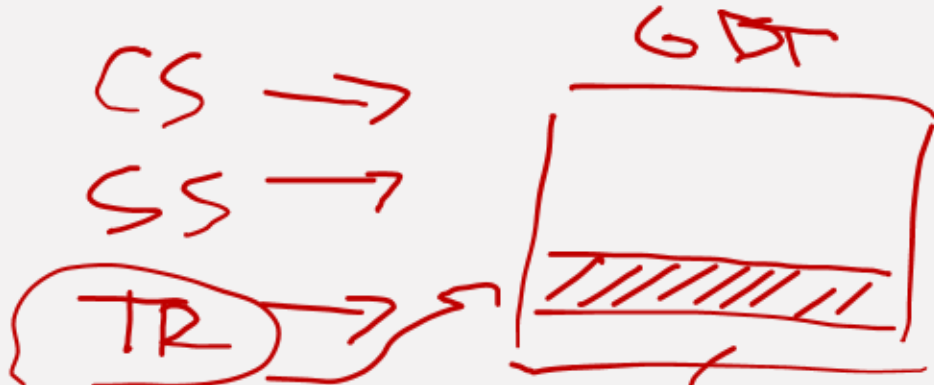
The manual requires 4-KiB alignment and recommends the memory type reported by `IA32_VMX_BASIC`, normally write-back cacheable memory. Uncacheable mappings can make transitions substantially slower.

04

The guest-state area

Enough state to resume precisely

STATE FAMILY	REPRESENTATIVE FIELDS	WHY HARDWARE NEEDS THEM
Execution mode and translation	<u>CR0</u> , <u>CR3</u> , <u>CR4</u>	They select protection, paging, address-space, and architectural features that determine how the next instruction is interpreted.
Debug environment	DR7, IA32_DEBUGCTL, pending debug exceptions	Breakpoint configuration and deferred debug delivery must remain guest-specific.
Continuation point	<u>RIP</u> , <u>RSP</u> , <u>RFLAGS</u>	They define where execution resumes, its stack, and its control/status flags.
Segmentation	CS, SS, DS, ES, FS, GS, LDTR, TR – selector/base/limit/access rights	The visible selector alone does not capture the hidden descriptor state actually used by the processor.
Descriptor tables	<u>GDTR</u> and <u>IDTR</u> base/limit	They locate and bound the guest's descriptor and interrupt tables.
<u>Mode-coupled MSRs</u>	IA32_EFER, IA32_PAT, <u>SYSENTER</u> state, CET/FRED state	These values change instruction semantics, memory typing, entry paths, protection, or tracing.
<u>Non-register execution state</u>	<u>activity</u> , <u>interruptibility</u> , <u>VMCS link</u> , <u>timer</u> , <u>cached PDPTes</u> , <u>PML index</u>	These describe what may happen at the next boundary even though no ordinary instruction reads them as one register.



Why RIP, RSP, and RFLAGS, but not every GPR?

— { P ≠ RAX
VMRESUME

These three are required to establish an execution context at a boundary. The VMCS has no fields for general-purpose registers such as RAX, RBX, and RCX.

DESIGN READING

GPRs need not be interpreted to establish the protected guest/host environment. A VM-exit assembly stub can save them immediately into the software vCPU structure before clobbering them.

AND A SECOND BENEFIT

Keeping that policy in software lets each hypervisor choose its own frame and calling convention rather than forcing a fixed hardware save area.

IMPLEMENTATION CONSEQUENCE

The VMCS is not a complete vCPU snapshot. A migratable or suspendable vCPU needs both VMCS-managed state and software-managed state.

What "mode-coupled" means

An MSR is mode-coupled when it participates directly in the execution environment being installed at VM entry. Letting a host value accidentally remain active could change the meaning of guest instructions or memory accesses before the hypervisor gets another chance to intervene.

Dedicated fields are not evidence that these are the only MSRs a guest may use. They are the MSRs for which Intel provides tightly defined transition handling. Others can be intercepted and emulated, or switched with the VM-entry/VM-exit MSR lists.

Dedicated guest fields exist for IA32_DEBUGCTL and the three SYSENTER MSRs, plus capability-dependent fields for IA32_PERF_GLOBAL_CTRL, IA32_PAT, IA32_EFER, IA32_BNDCFGS, IA32_RTIT_CTL, IA32_LBR_CTL, CET state, IA32_PKRS, FRED state, and IA32_SPEC_CTRL.

EXAMPLE

WHAT IT CHANGES

<u>IA32_EFER</u>	Long-mode, <u>execute-disable</u> , and <u>system-call semantics</u>
IA32_PAT	How <u>page-table memory types</u> are interpreted
<u>SYSENTER MSRs</u>	The <u>privilege-entry target selected by SYSENTER</u>
IA32_DEBUGCTL IA32_LBR_CTL IA32_RTIT_CTL	Debugging, branch recording, and tracing behavior
<u>CET state and SSP</u>	Control-flow enforcement and shadow-stack behavior
<u>IA32_PKRS</u> <u>uPK</u>	Supervisor protection-key permissions
FRED state	Event-delivery stacks and entry configuration
IA32_SPEC_CTRL	Speculative-execution policy

Why segment registers need four fields

This looks redundant only if a segment register is imagined as its visible selector. Loading a segment register also fills a hidden **descriptor cache** with base, limit, type, privilege, present, granularity, and mode information. That cached state can legitimately differ from the descriptor currently stored in the GDT or LDT: software may load the segment, then modify the table entry.

If VM entry re-read the descriptor from memory, it could reconstruct a different state from the one that existed at VM exit.
VM exit.

SELECTOR 16 bits Guest-visible name and privilege bits.	BASE 64 bits on Intel 64 Cached address added during translation where applicable.	LIMIT 32 bits, byte-granular Cached effective byte limit used for bounds checks.	ACCESS RIGHTS 32 bits Cached type, privilege, presence, mode, size, and granularity semantics.
--	---	---	---

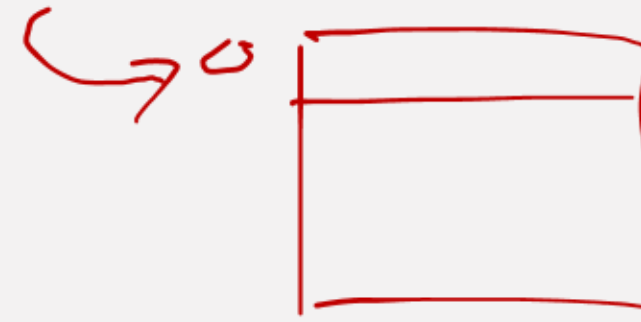
Descriptor-like, but not a descriptor

Bits 0:7, 12:15 Correspond to descriptor attributes: type, S, DPL, P, AVL, L, D/B, and G.

Bits 11:8 Reserved — even though the corresponding corresponding descriptor bits hold the upper upper four encoded limit bits.

Bit 16 Adds an **unusable** state.

The VMCS already stores the expanded effective limit in bytes as a separate 32-bit field. Repeating the descriptor's upper limit nibble would create two sources of truth. The G bit remains because it is part of architectural segment state and validation.



THE UNUSABLE BIT

A null selector typically makes a segment unusable, but ordinary ordinary access-right bits have no single descriptor encoding that that means "this segment register currently has no usable cached cached descriptor."

The explicit bit lets VM entry reproduce that processor state without inventing a descriptor.

ON 64-BIT BASE WIDTH

VMCS base fields are 64 bits including CS, SS, DS, and ES, though only 32 bits are architecturally defined for those bases. Intel states the width but gives no motive. Follow the VM-entry checks rather than treating field width as proof that every bit is usable.

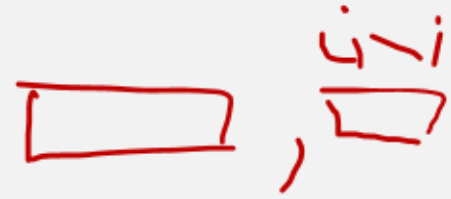


Table pointers versus descriptor selectors

GDTR AND IDTR — BASE AND LIMIT ONLY

Neither selects a descriptor, so neither has selector, type, DPL, or present fields. Each simply identifies a memory table by where it begins and the last valid byte offset, so the table size is limit + 1.

The base must span the architectural linear-address width because a descriptor table can reside anywhere permitted in the address space.

LDTR AND TR — ALL FOUR FIELDS

These *do* select descriptors: an LDT descriptor and a TSS descriptor. Like ordinary segment registers, they have visible selectors and hidden cached descriptor state.

That is why their VMCS representation includes selector, base, limit, and access rights rather than just base and limit.

On the 32-bit limit container. The architectural GDTR/IDTR limit is only 16 bits. VMX places it in a 32-bit field and VM entry requires bits 31:16 to be zero. Only the low 16 bits carry architectural information; the high 16 are storage width, not extra table size.

If a register is just a table pointer and bound, base plus limit is enough. If it selects a descriptor whose decoded state is cached, the VMCS must preserve both the selector and that cache.

Activity state: what should the CPU do next?

VALUE	STATE	MEANING
0	Active 	<u>Execute instructions normally</u>
1	HLT 	Wait for an event that wakes a halted processor
2	<u>Shutdown</u>	The virtual CPU has entered shutdown, commonly after a triple fault
3	<u>Wait-for-SIPI</u>	An application processor is waiting for a startup IPI

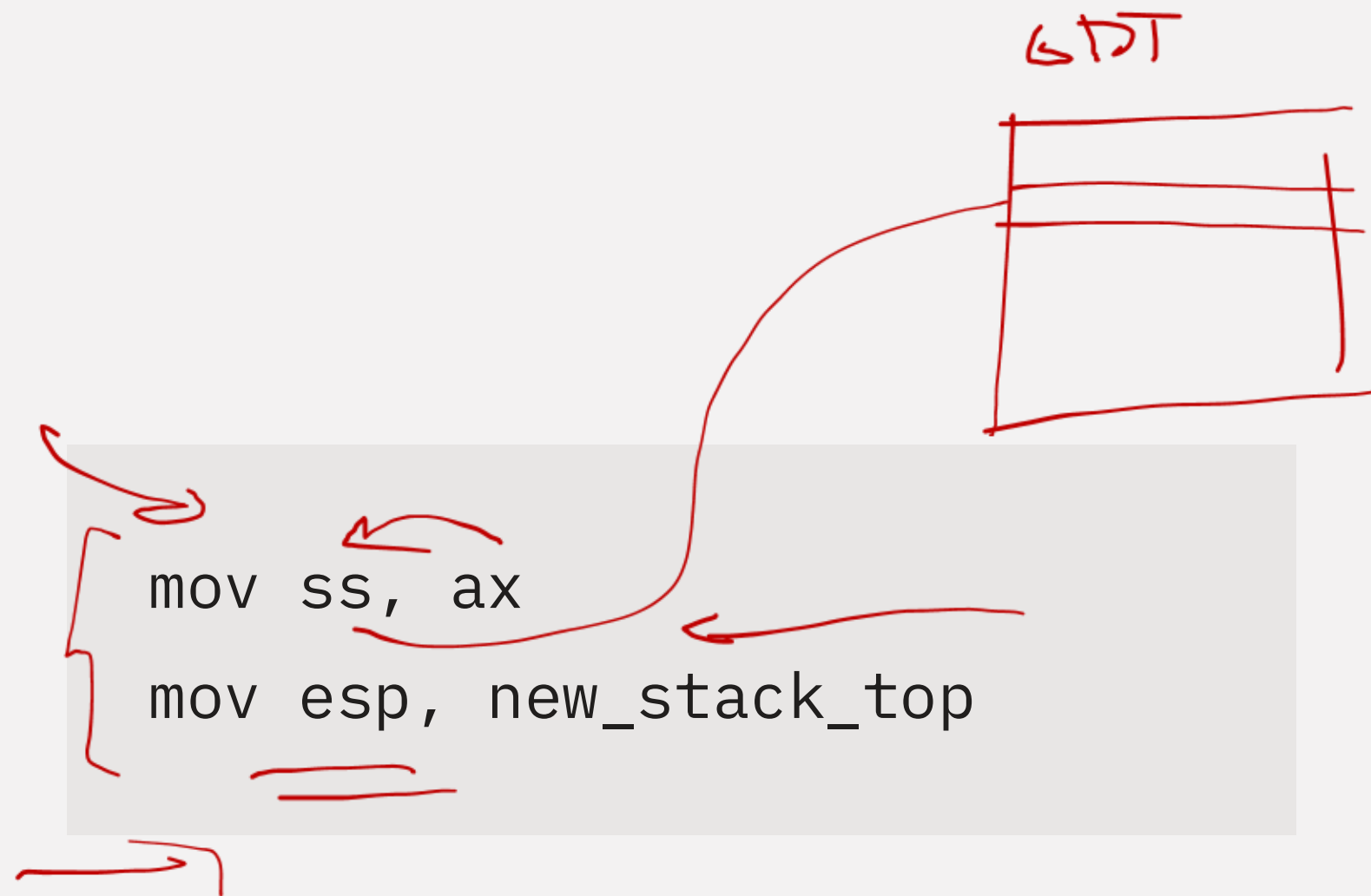
Saving only RIP cannot tell the processor whether it should execute that instruction now, sleep until an interrupt, or wait for startup. Supported inactive states are capability-reported because not every processor generation must implement every state.

Interruptibility state: why a durable register is insufficient

BIT	CONDITION	WHY IT MUST BE SAVED
0	Blocking by STI	STI can set <u>RFLAGS.IF</u> while still deferring interrupt recognition across the next instruction boundary. IF=1 alone loses that one-instruction shadow.
1	Blocking by MOV SS/POP SS	These instructions suppress certain interrupts, NMIs, and debug exceptions across the following boundary, historically allowing software to update the stack selector and stack pointer as a pair.
2	Blocking by SMI	While in SMM, further SMIs are blocked. That condition is not an ordinary bit in the guest's general register file.
3	Blocking by NMI	Delivering an NMI blocks another NMI until IRET; with virtual NMIs, the bit instead records the guest's virtual-NMI readiness.
4	Enclave interruption	Records that the exit occurred in, or incident to an event associated with, enclave execution.

Resuming with the right RFLAGS.IF but the wrong interrupt shadow changes architectural behavior. An interrupt could be delivered one instruction too early, or remain blocked too long.

A worked case: the stack switch



An interrupt between those two instructions could otherwise observe a new stack segment with the old stack pointer. The temporary `MOV SS` blocking state makes that sequence architecturally usable.

A VM exit between the instructions must preserve the shadow so resumption does not introduce an interrupt at a boundary where bare hardware would not.

THE GENERAL RULE

Virtualization must preserve not only register values, but also **causal history at an architectural boundary**.

Control-gated state

FIELD	STATE REPRESENTED
<u>VMX-preemption timer</u>	Remaining hardware-enforced guest time slice
PDPTE0–PDPTE3	 Cached page-directory-pointer entries used by PAE paging
Guest interrupt status	
<u>PML index</u>	
Guest deadline	
<u>VMCS link pointer</u>	The shadow VMCS used by non-root VMREAD/VMWRITE ; otherwise set to all ones

WHY GATING MATTERS

A hypervisor need not initialize data that the active controls will not use. New acceleration features can add state without forcing every older or minimal hypervisor to understand and populate it.

PENDING DEBUG EXCEPTIONS

The processor may recognize a debug condition but defer its delivery. The VMCS preserves that pending condition so a transition does not erase, duplicate, or reorder the guest's debug event.

Two CPUs with identical visible registers may still behave differently on the next instruction or event boundary. That difference is precisely this kind of non-register execution state.

05

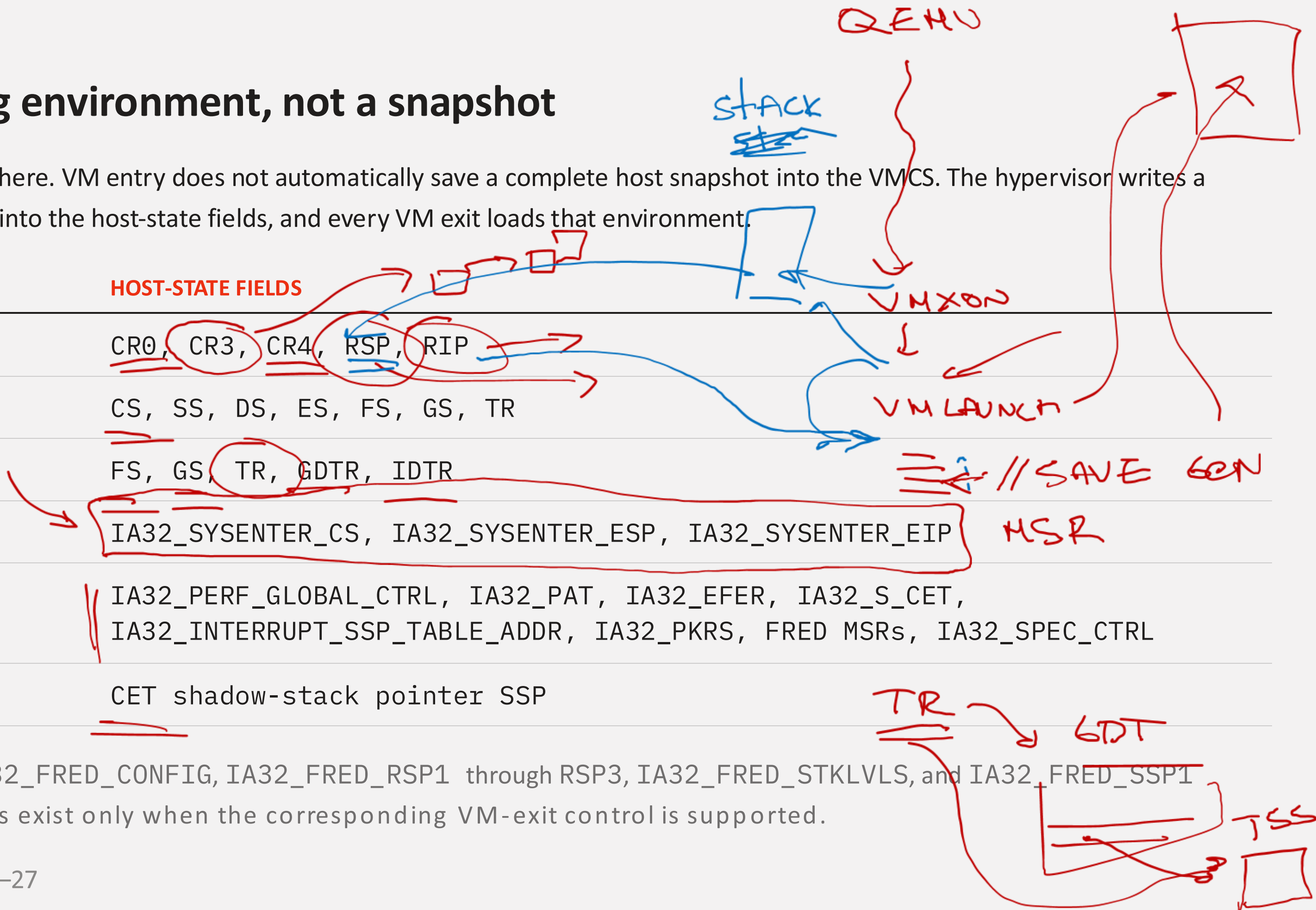
The host-state area

A prepared landing environment, not a snapshot

The word *restore* is misleading here. VM entry does not automatically save a complete host snapshot into the VMCS. The hypervisor writes a prepared landing environment into the host-state fields, and every VM exit loads that environment.

FAMILY	HOST-STATE FIELDS
Control and continuation	<u>CR0</u> , <u>CR3</u> , <u>CR4</u> , <u>RSP</u> , <u>RIP</u>
Segment selectors	<u>CS</u> , <u>SS</u> , <u>DS</u> , <u>ES</u> , <u>FS</u> , <u>GS</u> , <u>TR</u>
Bases	<u>FS</u> , <u>GS</u> , <u>TR</u> , <u>GDTR</u> , <u>IDTR</u>
Always-represented MSRs	<u>IA32_SYSENTER_CS</u> , <u>IA32_SYSENTER_ESP</u> , <u>IA32_SYSENTER_EIP</u>
Control-gated state	IA32_PERF_GLOBAL_CTRL, IA32_PAT, IA32_EFER, IA32_S_CET, IA32_INTERRUPT_SSP_TABLE_ADDR, IA32_PKRS, FRED MSRs, IA32_SPEC_CTRL
Control-gated register	<u>CET shadow-stack pointer SSP</u>

The FRED group consists of IA32_FRED_CONFIG, IA32_FRED_RSP1 through RSP3, IA32_FRED_STKLVLS, and IA32_FRED_SSP1 through SSP3. Optional fields exist only when the corresponding VM-exit control is supported.



The state needed before the first host instruction

CR3	Must select host page tables before the processor fetches from HOST_RIP.
CR0, CR4, EFER	Establish the interpretation and protection rules under which that fetch occurs.
<u>HOST_RIP</u>	Identifies the exit trampoline.
HOST_RSP	Provides a safe host stack before the trampoline executes PUSH, CALL, or takes an event.
CS	Together with the synthesized segment cache, establishes ring-0 code semantics.

Leaving guest CR3 active while fetching the host handler could fetch from a guest-controlled mapping, or fault hypervisor has executed even one instruction.

Leaving guest RSP active would make the first push overwrite guest memory.

No software prologue can repair state that it needs in order to run that prologue.

And what makes the machine safe immediately after

-
- `FS.base` and `GS.base` commonly address host thread-local or per-CPU data.
-
- `TR`, `GDTR`, and `IDTR` support segmentation and event delivery. `RFLAGS.IF=0` blocks ordinary maskable interrupts, interrupts, but exceptions, NMIs, and machine-check conditions still make valid event infrastructure important.
-
- `IA32_PKRS` affects whether supervisor accesses to pages are permitted.
-
- `IA32_PAT` affects the memory types used for host accesses.
-
- `CET`, `SSP`, and `FRED` state affect protected control transfers and event delivery.
-
- `IA32_SPEC_CTRL` applies the host's speculative-execution execution policy before host instructions run.
-

WHY SYSENTER AND PERF STATE ARE HERE AT ALL

They are not strictly required to fetch the first instruction. They are hardware-managed because they are common, transition-sensitive state for which atomic loading avoids repeated software boilerplate. The boundary is therefore not simply "impossible in software" versus "possible in software"; it also reflects security, frequency, and transition cost.

Segment state is deliberately normalized

The host area does not contain guest-style base, limit, and access-right fields for every segment. VM exit constructs a conventional ring-0 environment.

- CS, SS, DS, and ES bases are fixed or cleared rather than loaded from fields.
- Usable code/data segment limits are set to fixed large values.
- CS is constructed as a present ring-0 code segment; usable data segments as present ring-0 data segments.
- TR is constructed as a busy 32-bit TSS segment with a fixed limit; LDTR is cleared and marked unusable.
- GDTR and IDTR bases come from the VMCS, while both limits become FFFFH.

- NMI ← // ?

WHAT HAPPENS TO HOST RFLAGS?

There is no HOST_RFLAGS field. On VM exit the processor synthesizes:

RFLAGS = 000000000000000000002H

IF=0 — ordinary maskable interrupts initially remain disabled

DF=0 — string operations proceed toward higher addresses

TF=0 — the handler does not inherit single-step behavior

Arithmetic condition codes are cleared; the guest's STI/MOV SS shadows are not inherited.

This is possible because VM exit does not resume an arbitrary interrupted host instruction. It starts a new handler at a known entry point, which can execute STI later.

Other state established without host fields

STATE	VM-EXIT RESULT
DR7	400H
IA32_DEBUGCTL	Cleared
LDTR	Selector zero and unusable
Activity state	Active
STI and MOV-SS blocking	Cleared
Pending debug exceptions	Cleared ?
NMI blocking	Determined by how the VM exit occurred

CONTROL-SELECTED ON TOP

Additional controls can clear state such as UINV, IA32_BNDCFGS, and IA32_RTIT_CTL.

Host address-space-size and related rules determine mode-coupled pieces such as CS.L , EFER long-mode bits, and adjustments to CR4.

What the exit trampoline still saves in software

General-purpose registers other than the hardware-loaded RSP are not replaced with host values. Immediately after exit, RAX, RBX, RCX, R8–R15, and the others normally still contain guest values.



```
vmexit_entry:
```

```
    push rax
```

```
    push rbx
```

```
    push rcx
```

```
    ; save the rest of the
```

```
    ; guest register frame
```



The host's earlier general-purpose values were not secretly saved in the VMCS either. A vCPU-run wrapper preserves whatever host calling-convention state must survive, arranges the exit stack, and later reconstructs the software return path.

NOT SWITCHED EITHER

Floating-point, x87, SSE, AVX, and other XSAVE state are not automatically switched by VMX. The hypervisor or host OS manages them using an eager, lazy, or ownership-based policy.

THE DESIGN BOUNDARY

Hardware establishes the state needed to reach a safe, self-consistent host landing point. Software manages state that can wait until the exit trampoline is executing, especially large or policy-dependent state.

06

Controls, transitions, and exit information

Policy encoded for the fast path

in, out

PIN-BASED

Asynchronous events

External interrupts, NMIs, and the arrival policy for events the guest did not cause.

PROCESSOR-BASED

Instruction behavior

Mainly synchronous instruction interception, plus the switches that enable acceleration facilities.

ASSOCIATED FIELDS

Per-resource policy

Addresses, masks, thresholds, identifiers, and bitmaps that the enabled controls consume.

Asynchronous arrival policy and instruction semantics are different hardware paths. The grouping also allows later secondary and tertiary control vectors to be activated only when supported.

65K

PC

Exception bitmap: one bit per architectural exception

A 32-bit bitmap selects which exception vectors cause exits.

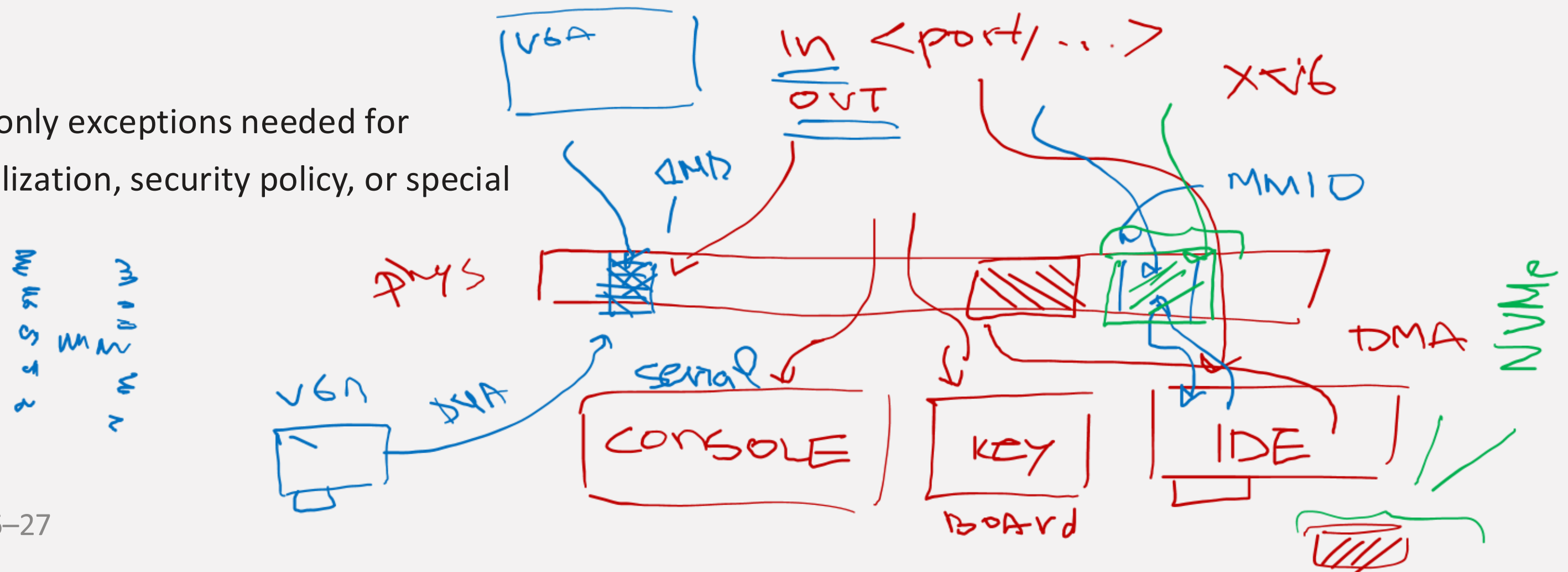
For page faults, error-code mask/match fields allow finer filtering.

Most exceptions should often stay inside the guest; the guest OS is supposed to handle its own divide errors and ordinary page faults.

The hypervisor intercepts only exceptions needed for debugging, memory virtualization, security policy, or special emulation.

WHY PAGE FAULTS GET EXTRA FILTERING

Different error-code classes may belong to the guest or to a shadow-paging mechanism. A single interception bit for `#PF` would force the hypervisor to see every guest page fault in order to find the few that are its own.



I/O bitmaps: why exactly two 4-KiB pages?

x86 has 65,536 I/O-port numbers. One interception bit per port requires 65,536 bits — exactly 8 KiB.

bitmap A

0000H–7FFFH

4 KiB · 32,768 ports



bitmap B

8000H–FFFFH

4 KiB · 32,768 ports

A set bit means an access to that port causes a VM exit.

The two aligned pages are simply a dense, constant-time policy table for the complete port namespace. More selective than unconditional I/O exiting, and far cheaper than a variable-length rule search in hardware.

MSR bitmaps: why four 1-KiB regions?

SUB-BITMAP BITMAP	MSR RANGE	OPERATION
0	000000000H-00001FFFFH	Read
1	C00000000H-C0001FFFFH	Read
2	000000000H-00001FFFFH	Write
3	C00000000H-C0001FFFFH	Write

Each covered range contains 8,192 MSR indices — 8,192 bits, or 1 KiB. 1 KiB. Accesses outside the covered ranges exit when MSR bitmaps are active.

WHY READ AND WRITE ARE SEPARATE

A value may be safe to reveal but unsafe to modify.

WHY NOT ONE CONTIGUOUS WINDOW

The bitmap covers 16K distinct indices, but not the first contiguous 16K: 8K low indices and 8K beginning at C00000000H . The high window contains extended system MSRs such as IA32_EFER at C0000080H and the 64-bit syscall MSRs.

THE ALTERNATIVE

One bit per possible 32-bit index would need 512 MiB, or 1 GiB for GiB for separate read and write policies. Most of that table would describe unimplemented holes.

Locating a policy bit

A LOW MSR

The bit number is simply the index. `IA32_TSC` is index `10H` , so its policy is bit 16: byte 2, bit 0. The read bit lives in the first sub-bitmap; the corresponding write bit lives at byte `2048 + 2`.

```
IA32_TSC  = 10H  → bit 16
  read   : byte    2, bit 0
  write  : byte 2050, bit 0
```

A HIGH MSR

First subtract `C00000000H`. `IA32_EFER` becomes offset `80H` , so its bit is byte 16, bit 0 within each high-range sub-bitmap.

```
IA32_EFER = C00000080H
  - C00000000H      → 80H
  read   : byte 1040, bit 0
```

OUTSIDE THE WINDOWS, EXITING IS THE CONSERVATIVE DEFAULT

That protects model-specific or package-wide machinery and provides forward compatibility: an MSR introduced by a future processor does not silently become guest-accessible to an older hypervisor. If the *use MSR bitmaps* control is disabled, all guest `RDMSR` and `WRMSR` instructions exit.

A clear bit means native access, not emulation

BIT	RESULT
0	The logical processor performs the real RDMSR or WRMSR
1	The instruction causes a VM exit before the access completes

The bitmap is an interception mechanism; it does not create a private bank of arbitrary MSRs for every VM. On the direct path, ordinary architectural validation and side effects still apply. If the index is unimplemented or the value violates that MSR's rules, the processor produces the normal exception, commonly `#GP`, subject to the VM's exception-interception policy.

If the bit is one, the hypervisor receives the index in `ECX` and, for `WRMSR`, the attempted value in `EDX:EAX`. It can validate the value, store a purely virtual copy, update hardware selectively, advance guest `RIP`, or inject `#GP`. In that path the hypervisor is emulating the access.

A worked case: IA32_LSTAR

If the write bit for the 64-bit SYSCALL target is clear, a guest WRMSR changes the real logical processor's LSTAR value, and a subsequent guest SYSCALL uses it natively. The hypervisor preserves isolation by context-switching the MSR.

```
VM entry:      load guest LSTAR from
                an MSR-load list
guest WRMSR:   real LSTAR becomes the
                guest's new value
VM exit store: save that value back to
                guest software state
VM exit load:  load the host's LSTAR
```

WHEN DIRECT ACCESS IS APPROPRIATE

Only when the MSR is safe for the guest while it runs, and can be switched or otherwise isolated at the boundary.

WHAT STAYS INTERCEPTED

MSRs with package-wide effects, physical-address information, tracing controls, power policy, or irreversible side effects.

AND A THIRD CASE

State such as virtual time or parts of the APIC interface receives receives purpose-built hardware virtualization, rather than either either ordinary pass-through or a full exit.

TSC offset and multiplier: virtual time without trapping every read

TSC offsetting adds a per-vCPU offset to the time-stamp value returned to the guest. TSC scaling applies a multiplier before the offset when supported. Doing the transformation in hardware lets RDTSC, RDTSCP, and selected TSC MSR reads remain on the native path.

```
guest_TSC =
  ((physical_TSC * multiplier) >> 48)
  + offset
```

2^{48} represents a multiplier of 1.0. Near 2^{47} gives half-speed virtual time; near 2^{49} gives double speed.



WHY THE OFFSET EXISTS

If a VM is paused while the physical TSC advances by five seconds, the hypervisor can adjust the offset so the first post-resume guest timestamp follows the last pre-pause timestamp without a five-second jump.

WHY THE MULTIPLIER EXISTS

If migration moves the vCPU to a machine whose TSC has a different origin or rate, the offset aligns the origin while the multiplier compensates for the rate difference.

Without hardware transformation, every frequent timestamp read would need a VM exit, software arithmetic, and emulation of the result.

Guest-host masks and read shadows: two views of one register

mask bit = 0

The guest owns it

Reads and allowed writes use the actual control-register bit. No exit for that bit.

mask bit = 1

The host owns it

Guest reads obtain the read-shadow bit, and an attempt to write a different value causes a VM exit.

Control registers mix harmless guest policy with machine-critical state. A single "exit on every control-register access" switch would be expensive, while direct access to every bit would be unsafe. Per-bit ownership lets hardware execute ordinary accesses and involve the hypervisor only when the guest tries to cross the chosen boundary.

EXAMPLE · HIDING CR4.VMXE

actual CR4.VMXE	= 1	
guest-host mask bit	= 1	host owns it
read-shadow bit	= 0	guest sees 0

A guest read of CR4 reports VMXE as zero. Keeping it zero is consistent with its virtual view and need not change the hardware bit. An attempt to write one crosses the policy boundary and causes an exit.

Similarly, a shadow-paging hypervisor may need real CR0.PG=1 while presenting a guest state in which paging is logically disabled.

CR3-target values: a small whitelist for hot address spaces

- If *CR3-load exiting* is 0, MOV to CR3 does not exit for this reason.
- If it is 1, the source value is compared with the first n CR3-target values, where n is the CR3-target count.
- A match avoids the exit. With count zero, every MOV to CR3 exits.
- The implementation described provides four target slots; software checks IA32_VMX_MISC rather than assuming the number.

```
CR3-load exiting = 1
```

```
new CR3 matches a target?
```

```
yes → load CR3 directly
```

```
no → VM exit for policy
```

Operating systems change CR3 during address-space switches. A hypervisor using shadow page tables may need to intercept a new guest page-table root so it can select or construct the corresponding shadow root. But the guest often cycles through a small set of hot roots — the current process and a kernel-related address space. Targets let those known values pass without paying for an exit every time.

WHY ONLY A SMALL SET · DESIGN READING

A tiny parallel comparison is cheap hardware and captures locality without turning the VMCS into a large address-space cache. The capability mechanism leaves room for a different number on future processors.

With EPT, a guest CR3 change does not by itself change the guest-physical-to-host-physical mapping, so a modern hypervisor can often set *CR3-load exiting* to zero and make targets irrelevant.

Scheduling intent, encoded as fields

INTERRUPT-WINDOW AND NMI-WINDOW EXITING

Notify me when delivery becomes possible

Sometimes the hypervisor has a virtual interrupt ready, but the guest currently blocks it. Polling the guest after every instruction would be absurd. Window exiting asks the processor to exit precisely when the guest reaches a boundary at which the event can be delivered.

It converts a "wait until condition" problem into a hardware notification. The guest runs freely while blocked; the hypervisor regains control at the first useful moment.

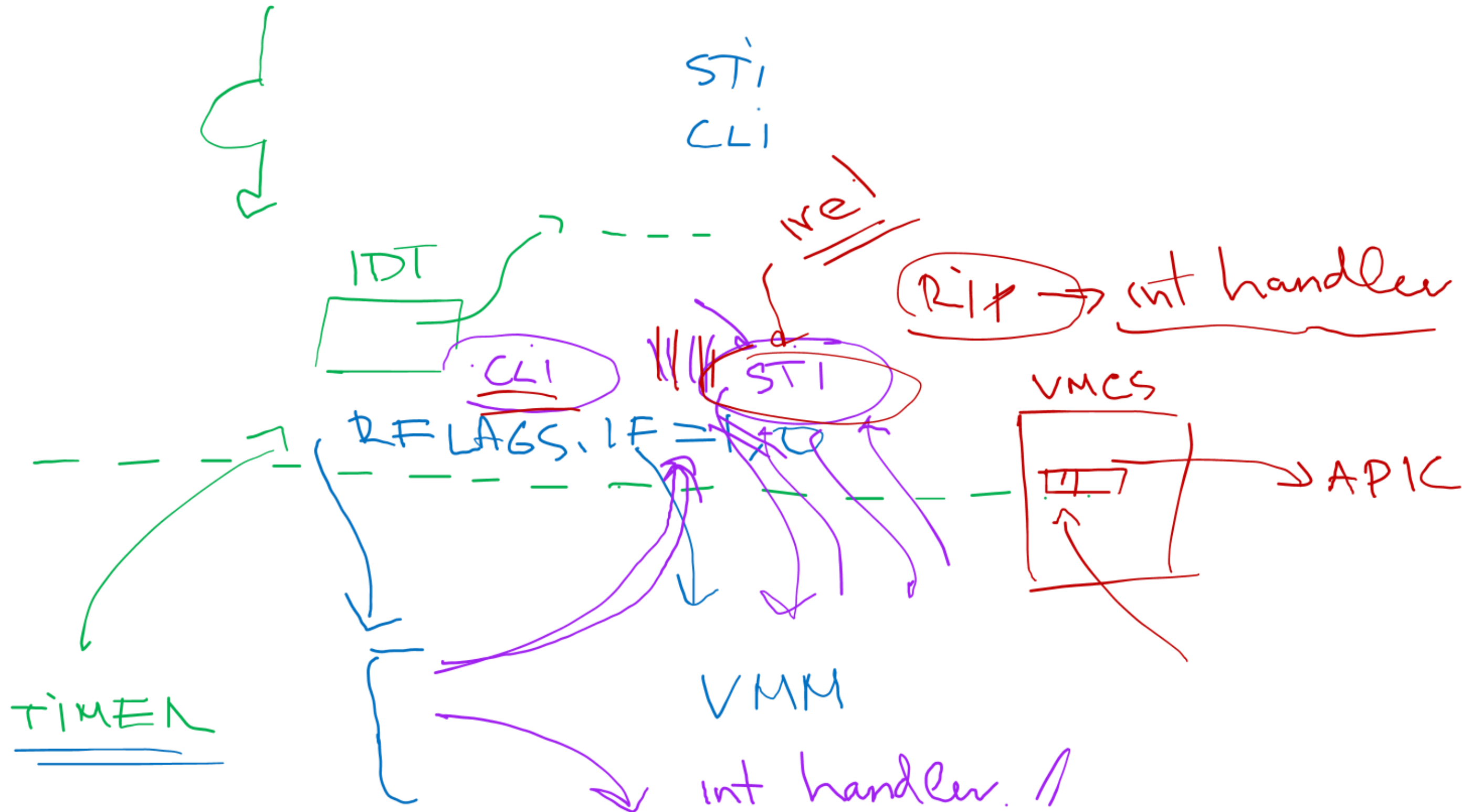
PREEMPTION TIMER AND PAUSE-LOOP EXITING

Bound the guest's run, and notice a spin

The preemption timer bounds how long a guest runs before an exit, independent of guest cooperation. PAUSE-loop exiting uses a gap and a window to distinguish isolated PAUSE instructions from a sustained spin loop.

- A short spin may be cheaper than descheduling.
- A long spin wastes a physical CPU and should return control to the scheduler.
- A time bound ensures the hypervisor eventually regains control even if the guest masks interrupts.

Options
- Guest owns it



Why entry and exit controls are separate

Execution controls govern the guest while it runs. Entry and exit controls govern the boundary itself. Examples include whether:

- debug controls, IA32_PAT, IA32_EFER , CET, FRED, protection-key state, and other supported components are loaded or saved;
- the guest enters IA-32e mode;
- the host exit target uses a 64-bit address space;
- an external interrupt is acknowledged during exit;
- selected MSRs are loaded or stored through list areas;
- an event is injected as entry completes.

WHY DIRECTION-SPECIFIC

Guest and host do not necessarily need symmetric treatment. A hypervisor may want to save a guest value, load a different host value, clear sensitive state on exit, or leave a component unchanged for performance. Separate controls express those choices without forcing every transition to copy every possible register.

DEDICATED FIELDS PLUS MSR LISTS

Frequently transition-coupled MSRs receive dedicated VMCS fields and controls. Additional MSRs are handled through count-and-address list areas containing 16-byte entries.

Dedicated fields make common, mode-sensitive state straightforward and validatable. Lists provide an extensible long tail without permanently enlarging the fixed VMCS interface for every MSR Intel may add.

Why injection belongs to VM entry

VM entry can conclude by delivering a selected interrupt, NMI, exception, or other supported event. The VMCS provides the vector, type, validity, error-code information, and — for software events — an instruction length used in constructing the guest-visible return point.

The processor can deliver the event only after the intended guest state is installed. Making injection part of the atomic entry sequence avoids a race in which guest instructions run before a pending virtual event is delivered.

Why exit-information fields are rich

An exit reason alone is rarely enough to emulate an operation. Chapter 27 defines fields for:

- basic exit reason and flags such as failed VM entry;
- exit qualification, whose format depends on the reason;
- guest linear and guest-physical addresses when applicable;
- exiting-event vector, type, and error information;
- original-event information when an exit interrupts event delivery;
- instruction length and instruction information for supported instruction exits;
- VM-instruction errors, which are distinct from VM exits.

WHY REPORT INSTRUCTION INFORMATION

The processor has already decoded the instruction and knows the addressing details that caused the exit. Reporting them avoids a second decode in the hypervisor, including difficult cases involving address size, segment selection, or an inaccessible instruction page.

WHY PRESERVE THE ORIGINAL EVENT

An exit can occur while the processor is attempting to deliver an interrupt or exception — for example, if event delivery itself encounters a protected memory access. If only the new exit were reported, the original event could be lost or injected twice.

Virtualization must preserve not only register values, but also causal history at an architectural boundary.

Why shadow VMCSs exist

L0

The real hypervisor

In VMX root operation. Owns the ordinary VMCS that actually governs non-root execution.

L1

A guest hypervisor

In VMX non-root operation. Receives a hardware-accelerated *view* of a VMCS.

L2

The nested guest

The guest that L1 believes it controls.

Without VMCS shadowing, every L1 VMREAD and VMWRITE would exit to L0. A shadow VMCS lets selected accesses execute directly. VMREAD and VMWRITE bitmaps let L0 decide which field encodings are safe to access without an exit.

THE SECURITY BOUNDARY

The shadow VMCS cannot itself be used for real VM entry. L0 is still responsible for validating and synchronizing the relevant shadow values into the ordinary VMCS before real L2 entry.

Why "allowed zero" and "allowed one" both matter

For the traditional 64-bit capability-MSR format:

low 32 bits Zero-settings. A low-half bit of 1 means the control cannot be zero, and is therefore required to be one.

high 32 bits One-settings. A high-half bit of 0 means the control cannot be one, and is therefore required to be zero.

```
uint32_t legal = desired;
legal |= (uint32_t)capability;
legal &= (uint32_t)(capability >> 32);
```

WHY BOTH HALVES

A processor may require an old default behavior, permit a feature optionally, or reserve a newer feature to zero. `IA32_VMX_BASIC` indicates when the "true" control MSRs should be used for complete reporting of older default-one controls.

Secondary and tertiary controls use their own documented capability formats and enabling controls. The formula above should not be copied blindly to every newer vector.

WHY THIS DESIGN AGES WELL

A control word is negotiated between software intent and hardware capability. Unknown bits remain safe, required legacy behavior is retained, and a new processor can add optional functionality without changing the basic VM-entry instruction.

A field-design checklist for reading Chapter 27

- 1 **Is it state, policy, or diagnostic information?** That identifies the VMCS group and when it is meaningful.
- 2 **Is it visible architectural state or hidden boundary state?** Segment caches and interrupt shadows explain many fields that appear redundant.
- 3 **Which control activates it?** Many addresses and optional fields are ignored when their feature is disabled.
- 4 **Why is the granularity a bit, bitmap, exact target, threshold, or list?** The representation usually matches the lookup the fast path must perform.
- 5 **Which side owns the value?** Guest-host masks, read shadows, and entry/exit direction encode ownership.
- 6 **What must survive migration?** Processor-cached VMCS data requires VMCLEAR ; software-managed GPRs require a separate vCPU frame.
- 7 **What must be validated before entry?** Reserved bits, canonical addresses, alignment, mode consistency, and control dependencies define the safety contract.

CLOSE

The design in one paragraph

VMX creates an execution environment in which a guest can keep its native privilege structure while the hypervisor retains final control. The VMCS is deliberately not a memory-mapped register dump. It is a versioned, opaque, per-vCPU contract divided into resumable state, interception policy, transition policy, and exit diagnostics. Fields exist when hardware must reproduce hidden architectural state, make a fast-path decision, or explain a transition without ambiguity.

Bitmaps handle large dense namespaces.

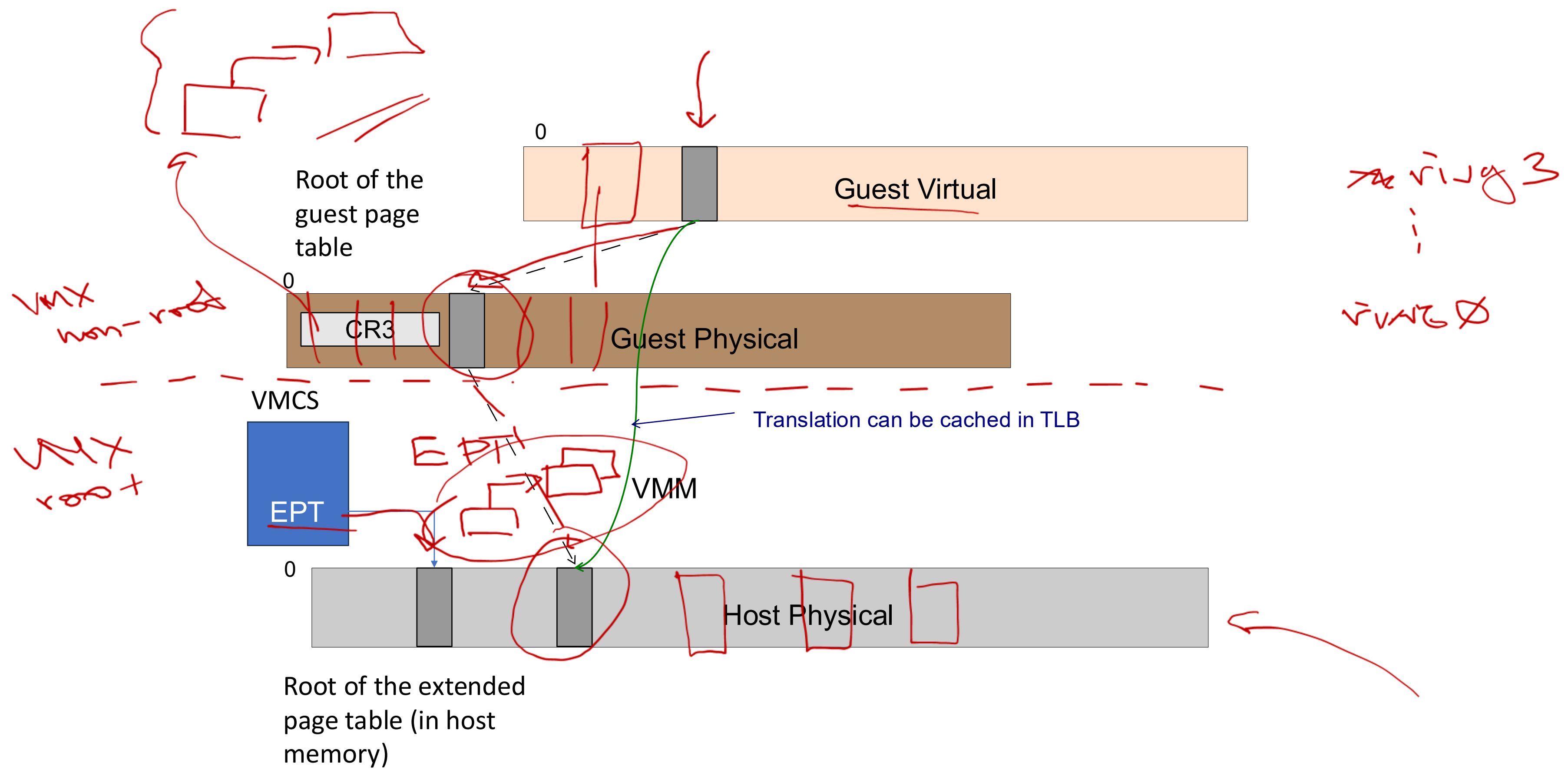
Masks divide ownership bit by bit.

Small target lists exempt hot exact values.

Pointers attach page-sized acceleration structures.

Entry and exit controls express asymmetric state changes.

Capability MSRs negotiate everything else.



Source and scope

Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, & (3A, 3B, 3C, & 3D): System Programming Guide. Order Number 325384-092US, June 092US, June 2026.

PRIMARY READING

26.2–26.5	VM architecture, root/non-root operation, lifecycle, <u>VMCS</u> introduction
26.6–26.8	Discovery, enablement, VMXON, and VMX restrictions
27.1–27.3	VMCS state, region format, and logical organization
27.4–27.5	<u>Guest-state and host-state areas</u>
27.6.1–27.6.27	<u>Execution controls and associated fields</u>
27.7–27.9	<u>Exit controls, entry controls, injection, and exit information</u>
27.10–27.11	<u>Shadow VMCSs, ownership rules, encodings, and initialization</u>
30.5	Exact state loaded, synthesized, or cleared on VM exit

WHERE TO GO NEXT

Chapters 28–33 define the run-time behavior and validation that consume these fields. Appendix A describes capability reporting, Appendix B lists VMCS field encodings, and Appendix C lists basic exit reasons.

ON THE DESIGN READINGS

Rationale marked as a *design reading* is engineering interpretation of the specified behavior, not a claim that Intel has documented that historical or microarchitectural motive.

Production hypervisor code must follow the manual's exact exact checks, capability rules, memory-ordering requirements, requirements, instruction pseudocode, and applicable processor processor errata.