Advanced x86: Virtualization with VT-x Part 2

David Weinstein
dweinst@insitusec.com

All materials are licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Hardware Assisted Virtualization

- Hardware provides the heavy lifting to deal with all these issues
- Host (dom0 or VMM) – whoever is there "first"
  - in charge of creating VM "control structures" (VMCSs)
- Before a guest runs, specify a number of events/state that cause VM exits
  - Think "bitmask" of interrupts to jump out to VMM land

Hardware Assisted Virtualization (2)

- Guest runs until
  - It does something that has been registered in data structures (i.e., VMCS) to exit out to VMM,
  - It explicitly calls the VMCALL instruction
- VMM can preempt guest regularly with on a timer
- VMM can virtualize access to guest's memory
  - "guest-physical" addresses
- Instructions can cause trap to the VMM

Basic Idea...

VMCS (intel)

VMCB Data Struct

Host instruction Stream

```
while (1) {
   // Do World Switch
   rAX = &VMCB
   VMLOAD(rAX)

   while (running_VMM) {
      VMRUN(rAX)

      switch (exitcode) {
      // handle intercept
      // within VMM context
   }

   VMSAVE(rAX)
}
```

VMXON

Guest instruction Stream

Intercepts

BluePill/HyperJacking Techniques

- Does not virtualize hardware
  - BP’d systems see same hardware before/after
- Early PoCs, BlackHat 2006
  - BluePill (Rutkowska/Tereshkin @ COSEINC) , AMD-v, Windows
  - Vitriol (Dino Dai Zovi @ Matasano), Intel VT-x, Mac OS X
- Early academic PoC
  - SubVirt, Samuel T. King et al

General Hardware VM Based Rootkit

- Virtual Machine Based Rootkit (VMBR)
- Start with CPL=0
- Allocate some unpaged physical memory
  - Ensure no linear mappings to VMM after guest entry
- Move running OS into VMCS
- Intercept access to hardware (IO ports, ...)
- **Communicate to hardware VM rootkit via sentinel instructions**

VMX introduces new x86 instructions

VMXON	Enable VMX
VMXOFF	Disable VMX
VMLAUNCH	Start/enter VM
VMRESUME	Re-enter VM
VMCLEAR	Null out/reinitialize VMCS
VMPTRLD	Load the current VMCS
VMPTRST	Store the current VMCS
VMREAD	Read values from VMCS
VMWRITE	Write values to VMCS
VMCALL	Exit virtual machine to VMM
VMFUNC	Invoke a VM function in VMM without exiting guest operation

High level VMX flow

- VMM will take these actions
  - Initially enter VMX mode using VMXON
  - Clear guest's VMCS using VMCLEAR
  - Load guest pointer using VMPTRLD
  - Write VMCS parameters using VMWRITE
  - Launch guest using VMLAUNCH
  - Guest exit (VMCALL or instruction, …)
  - Read guest-exit info using VMREAD
  - Maybe reenter guest using VMRESUME
  - Eventually leave VMX mode using VMXOFF

Introducing Chicken Syrup

- Your toy VMM, pieced together from various bits
- Frankenchicken?
- Windows 7 x64 Driver
- Based on Virtdbg
  - We’re building up to virtdbg’s feature set

MSRs and VMX capabilities

- MSRs used to identify capabilities of the hardware
- We'll refresh on how to access MSRs and talk about how each plays a role in implementing a VMM
- Appendix A goes into detail on each.
- These slides should have what you need, though.

Relevant VMX MSRs (1)

- IA32_VMX_BASIC (0x480)
  - Basic VMX information including revision, VMXON/VMCS region size, memory types and others.
- IA32_VMX_PINBASED_CTLS (0x481)
  - Allowed settings for pin-based VM execution controls.
  - When you see Pin, think *asynchronous* events/interrupts
- IA32_VMX_PROCBASED_CTLS (0x482)
  - Allowed settings for *primary* processor based VM execution controls. Things like exiting on specific instruction execution
- IA32_VMX_PROCBASED_CTLS2 (0x48B)
  - Allowed settings for *secondary* processor based VM execution controls.

IA32_ naming convention just means it's an architectural MSR, nothing to do with 32-bit specifically

Relevant VMX MSRs (2)

- IA32_VMX_EXIT_CTLS (0x483)
  - Allowed settings for VM Exit controls.
- IA32_VMX_ENTRY_CTLS (0x484)
  - Allowed settings for VM Entry controls.
- IA32_VMX_MISC (0x485)
  - Allowed settings for miscellaneous data, such as RDTSC options, unrestricted guest availability, activity state and others.

Relevant VMX MSRs (3)

- IA32_VMX_CR0_FIXED{0,1}   0x486, 0x487
  - Indicate the bits that are allowed to be 0 or to 1 in CR0 during VMX operation.
- IA32_VMX_CR4_FIXED{0,1}   0x488, 0x489
  - Same for CR4.
- IA32_VMX_VMCS_ENUM   0x48A
  - Enumeration helper for VMCS.
- IA32_VMX_EPT_VPID_CAP 0x48C
  - Provides information for VPIDs/EPT capabilities.

#define MSRs

```
#define MSR_IA32_FEATURE_CONTROL   0x03a
#define MSR_IA32_VMX_BASIC               0x480
#define MSR_IA32_VMX_PINBASED_CTLS          0x481
#define MSR_IA32_VMX_PROCBASED_CTLS       0x482
#define MSR_IA32_VMX_EXIT_CTLS              0x483
#define MSR_IA32_VMX_ENTRY_CTLS           0x484
#define MSR_IA32_VMX_CR0_FIXED0 0x486
#define MSR_IA32_VMX_CR0_FIXED1 0x487
#define MSR_IA32_VMX_CR4_FIXED0 0x488
#define MSR_IA32_VMX_CR4_FIXED1 0x489
```

MSR: IA32_FEATURE_CONTROL (index 0x03a)

- Controls the ability to turn VMX "on"
  - Usually controlled by the BIOS to enable/disable virtualization
- Gets set to 0 on CPU reset
- If not configured appropriately our VMX instructions will generate invalid opcode exceptions
- **Bit 0** is the lock bit. If 0, BIOS has locked us out of VMX
- **Bit 1** enables VMX in SMX operation.
  - Outside scope of this class
- **Bit 2** enables VMX outside SMX, *which we need*
- **In our case virtualization should already be turned "on" in system BIOS, but please verify this for yourself.**

IA32_FEATURE_CONTROL in C

```
typedef struct
_IA32_FEATURE_CONTROL_MSR
{
    unsigned Lock               :1;
    unsigned VmxonInSmx         :1;
    unsigned VmxonOutSmx        :1;
    unsigned Reserved2         :29;
    unsigned Reserved3         :32;
} IA32_FEATURE_CONTROL_MSR;
```

MSR: IA32_VMX_CR0_FIXED0/1 (index: 0x486, 0x487)

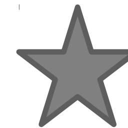
- Bit X in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in CR0_FIXED0 and 1 in CR0_FIXED1).
- If bit X is 1 in CR0_FIXED0, then that bit is also 1 in CR0_FIXED1

MSR: IA32_VMX_CR4_FIXED0/1 (index: 0x488, 0x489)

- Bit X in CR4 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in CR4_FIXED0 and 1 in CR4_FIXED1).
- If bit X is 1 in CR0_FIXED0, then that bit is also 1 in CR4_FIXED1

CR4 Typedef

```
typedef struct _CR4_REG {
    unsigned VME        :1;    // Virtual Mode Extensions
    unsigned PVI        :1;    // Protected-Mode Virtual Interrupts
    unsigned TSD        :1;    // Time Stamp Disable
    unsigned DE         :1;    // Debugging Extensions
    unsigned PSE        :1;    // Page Size Extensions
    unsigned PAE        :1;    // Physical Address Extension
    unsigned MCE        :1;    // Machine-Check Enable
    unsigned PGE        :1;    // Page Global Enable
    unsigned PCE        :1;    // Performance-Monitoring Counter Enable
    unsigned OSFXSR     :1;    // OS Support for FXSAVE/FXRSTOR
    unsigned OSXMMEXCPT :1;    // OS Support for Unmasked SIMD Floating-Point Exceptions
    unsigned Reserved1  :2;    //
    unsigned VMXE       :1;    // Virtual Machine Extensions Enabled
    unsigned Reserved2  :18;   //
} CR4_REG, *PCR4_REG;
```



Enabling and Entering the Matrix

- In addition to IA32_FEATURE_CONTROL…
- Before entering VMX operation, enable VMX by setting CR4.VMXE[bit 13] = 1
  - Or VMX instructions will also generate invalid-opcode exceptions
- VMX operation is then entered by executing the VMXON instruction

```
; Enable VMX by setting CR4.VMXE
MOV  eax, cr4
BTS    eax, 13
MOV  cr4, eax
```

VMXON

VMXON—Enter VMX Operation

Opcode	Instruction	Description
F3 0F C7 /6	VMXON m64	Enter VMX root operation.

Description

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.[1]

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

4. If IA32_VMX_BASIC[48] is read as 1, VMfailInvalid occurs if addr sets any bits in the range 63:32; see Appendix G.1.

VMXON vs VMCS region

- We will talk about a VMXON region and a VMCS
- The VMXON region is created **per logical processor** and used by it for VMX operations
- The VMCS region is created **per guest virtual cpu** and used both by the hypervisor and the processor.

4KB-aligned Address stated different ways

- Means 0x*****000 or 0x*************000
  - i.e., ends in hex 000
  - ADDR % 4096 = 0
  - remainder when dividing by 4096 is 0

- Some of our data structures will require this 4KB alignment condition.

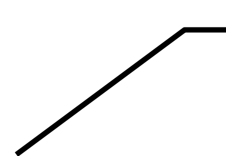
MSR: IA32_VMX_BASIC (index 0x480)

- Bits 31:0 contain the 32-bit VMCS revision identifier
- Bits 44:32 report the # of bytes to allocate for the VMXON/VMCS regions
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width. If the bit is 1, these addresses are limited to 32 bits.
  - This bit is always 0 for processors that support Intel 64 architecture.

```
XOR     ecx, ecx
MOV     ecx, 0480h
RDMSR
; result is now in edx:eax (i.e., 64 bits across two 32 bit registers)
```

IA32_VMX_BASIC as C struct

```
typedef struct _VMX_BASIC_MSR {
    unsigned RevId:32;
    unsigned szVmxOnRegion:12;
    unsigned ClearBit:1;
    unsigned Reserved:3;
    unsigned PhysicalWidth:1;
    unsigned DualMonitor:1;
    unsigned MemoryType:4;
    unsigned VmExitInformation:1;
    unsigned Reserved2:9;
} VMX_BASIC_MSR, *PVMX_BASIC_MSR;
```



0 – Uncacheable (UC)
6 – Write Back (WB)

Reading MSRs

In our amd64/amd64.asm:

```
_ReadMsr PROC
    xor    rax, rax
    rdmsr      ; MSR[ecx] --> edx:eax
      shl           rdx, 32
      or            rax, rdx
      ret ; don't forget to ret or you will mess up your stack
_ReadMsr ENDP
```

In our src/amd64.h:

```
ULONG64 _ReadMsr(ULONG32 reg);
```

In our src/vmx.c :

```
PVMX_BASIC_MSR pvmx;
ULONG64 msr;
msr = _ReadMsr(MSR_IA32_VMX_BASIC);
pvmx = (PVMX_BASIC_MSR)&msr;
```

Lab: VMXMSR driver

- Write a Windows driver to check the value of MSR index 0x480 – 0x48A, + FEATURE_CONTROL (0x03a)
- Use the C structs provided for MSR_IA32_VMX_BASIC/CR4 to dump a detailed description
- Also grab and dump the CR4 value and for convenience whether CR4.VMXE is set to 1
- Use DbgPrint's to print the values and Sysinternals DbgView to view the results.

Allocating VMXON Region in Windows Driver

In our src/vmx.c :

```
typedef LARGE_INTEGER PHYSICAL_ADDRESS, *PPHYSICAL_ADDRESS;
PVMX_BASIC_MSR pvmx;
PHYSICAL_ADDRESS pa;
PVOID va;
/* Read VMX_BASIC MSR into pvmx */

/* determine size from bits 44:32 of IA32_VMX_BASIC MSR or assume 4K … */
va = AllocateContiguousMemory(size);

/* check for null va, set VMX revision ID */
*(ULONG32 *)va = pvmx->RevId;
pa = MmGetPhysicalAddress(va)
…
/* set CR4.VMXE bit => */

_VmxOn(…)
```

Allocating VMXON Region in Windows Driver

```
; pure assembly implementation
.data
vmxon_ptr dq    ; initialized elsewhere
…
.code
MOV ecx, 0x480              ;  IA32_VMX_BASIC MSR
RDMSR                       ;
MOV   edx, [vmxon-ptr]  ;  load VMXON region
MOV  [edx], eax             ;  VMX revision id into offset 0 of VMXON region
```

PHYSICAL_ADDRESS

typedef LARGE_INTEGER PHYSICAL_ADDRESS, *PPHYSICAL_ADDRESS;

(LARGE_INTEGER is declared in ntddk.h)

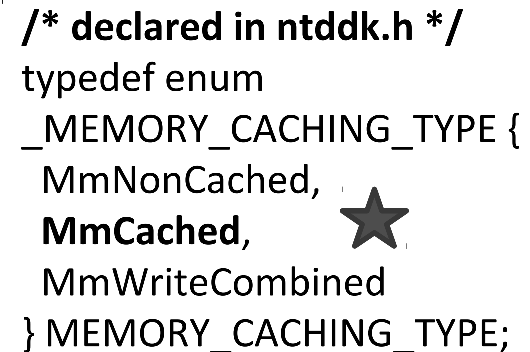
AllocateContiguousMemory

```
PVOID AllocateContiguousMemory(ULONG size)
{
    PVOID Address;
    PHYSICAL_ADDRESS l1, l2, l3;
    l1.QuadPart = 0;
    l2.QuadPart = -1;
    l3.QuadPart = 0x200000;
    Address = MmAllocateContiguousMemorySpecifyCache(size, l1, l2, l3,    MmCached);
    if (Address == NULL) {
        return NULL;
    }
    RtlZeroMemory(Address, size);
    return Address;
}
```

```
NTKERNELAPI PVOID
  MmAllocateContiguousMemorySpecifyCache(
    IN SIZE_T  NumberOfBytes,
    IN PHYSICAL_ADDRESS  LowestAcceptableAddress,
    IN PHYSICAL_ADDRESS  HighestAcceptableAddress,
    IN PHYSICAL_ADDRESS  BoundaryAddressMultiple  OPTIONAL,
    IN MEMORY_CACHING_TYPE  CacheType
    );
/* declared in ntddk.h */
```

MEMORY_CACHING_TYPE

- MmNonCached
  - The requested memory should not be cached by the processor.
- MmCached
  - The processor should cache the requested memory.
- MmWriteCombined
  - The requested memory should not be cached by the processor, but writes to the memory can be combined by the processor.



```
/* declared in ntddk.h */
typedef enum
_MEMORY_CACHING_TYPE {
  MmNonCached,
  MmCached,
  MmWriteCombined
} MEMORY_CACHING_TYPE;
```

Lab: VMXON!

- Expand on VMXMSR lab to allocate VMXON region, initialize it, and turn VMX on!
- Allocate contiguous regions for VMXON region based on appropriate size (hint IA32_VMX_BASIC)
- You'll need to make another assembly function with C prototype:
  - VOID _VmxOn(PHYSICAL_ADDRESS PA);

Lab Review

- CheckIfVMXIsSupported()
  - CPUID leaf 1, bit 5 in ecx = 1 ?
- CheckIfVMXIsEnabled()
  - Check CR4 bit 13 = 1 ? Set it if not.
  - Check IA32_FEATURE_CONTROL bit 2 = 1?
- SetupVMX()
  - ReadMsr IA32_VMX_BASIC for VMXON region size
  - And VMX revision ID
  - AllocateContiguousMemory(size)
  - Set Revision ID in VMXON region
  - Call VMXON with 64-bit <u>Physical</u> Address
- Download sample solution to VMXMSR and VMXON labs

Possible mistakes/gotchas

- Forget to RET in your assembly PROC
- Forget to set VMX Revision ID in VMXON region
- Make sure you use the right MSR index numbers
- x64 calling convention
- Multiple logical processors (have to loop on them and perform these steps on each to be kosher)
  - KeSetSystemAffinityThreadEx

Multi-processor virtualization

- Symmetric VMM is most common
  - Same effective VMM on all logical processors
  - It's what we've been talking about thus far
- Asymmetric configuration is possible though
  - i.e., VMMs with different VMX revision id, exit controls
  - The benefits might be to allow for migration of VMs across a cluster
  - Out of scope for this class. Perhaps a topic for an advanced class

VMM Design Considerations (1)

- Multi-processor
  - symmetric vs. asymmetric
  - Locking mechanisms to protect shared VMM data
  - Meta information about multiple VMCSs for example and state tracking
  - If your VMM is for debugging, info about task, etc…
  - Don’t forget we’re virtualizing on multiple logical cores

VMM Design Considerations (2)

- Also depends on your design goals
  - Are you Hyper-jacking? (i.e., for debug, bluepill)
  - Or hosting multiple guest OS types? Like VMware
  - Do you want to fully support guest VM self-debug?
  - Is your goal to do debugging on malware?
  - Does speed matter?
  - Are you going to migrate VMs to other hardware platforms? Are you an IaaS provider?

Multi-processor initialization

```
for (i = 0; i < KeNumberProcessors; i++)
  {
    OldAffinity = KeSetSystemAffinityThreadEx((KAFFINITY) (1 << i));
    OldIrql = KeRaiseIrqlToDpcLevel();
    _StartVirtualization();
    KeLowerIrql(OldIrql);
    KeRevertToUserAffinityThreadEx(OldAffinity);
  }
```

KeNumberProcessors

- It’s bound to the total number of logical processors
- Obsolete and shouldn’t be used anymore… but it still works
- “In Windows Server 2008, code that can determine the number of processors must use KeQueryActiveProcessors”
- Read the MSDN reference below on alternative

http://msdn.microsoft.com/en-us/library/windows/hardware/ff552975%28v=vs.85%29.aspx

KeSetSystemAffinityThreadEx

- Sets the system affinity of the current thread
  - Parameter is actually a **set** of possible processors
  - In our case we pick a specific one (i.e., 1 << i)
- Returns either the previous system affinity of the current thread, or zero to indicate that there was no previous system affinity
- Callers should save the return value and later pass this value to the KeRevertToUserAffinityThreadEx routine to restore the previous affinity mask.

KeRaiseIrqlToDpcLevel

- Raises the hardware priority to IRQL = DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.
- Caller should save the returned IRQL value and restore the original IRQL as quickly as possible by passing this returned IRQL in a subsequent call to KeLowerIrql

Managing Multiple VMXON regions

```
typedef struct _VIRT_CPU {
    PVOID Self,
    PVOID VMXON_va;
    PHYSICAL_ADDRESS VMXON_pa;
    PVOID VMCS_va;
    PHYSICAL_ADDRESS VMCS_pa;
    …
} VIRT_CPU, *PVIRT_CPU;
```

VMXOFF

VMXOFF—Leave VMX Operation

Opcode	Instruction	Description
0F 01 C4	VMXOFF	Leaves VMX operation.

Description

Takes the logical processor out of VMX operation, unblocks INIT signals, conditionally re-enables A20M, and clears any address-range monitoring.[1]

Wax on, wax off...

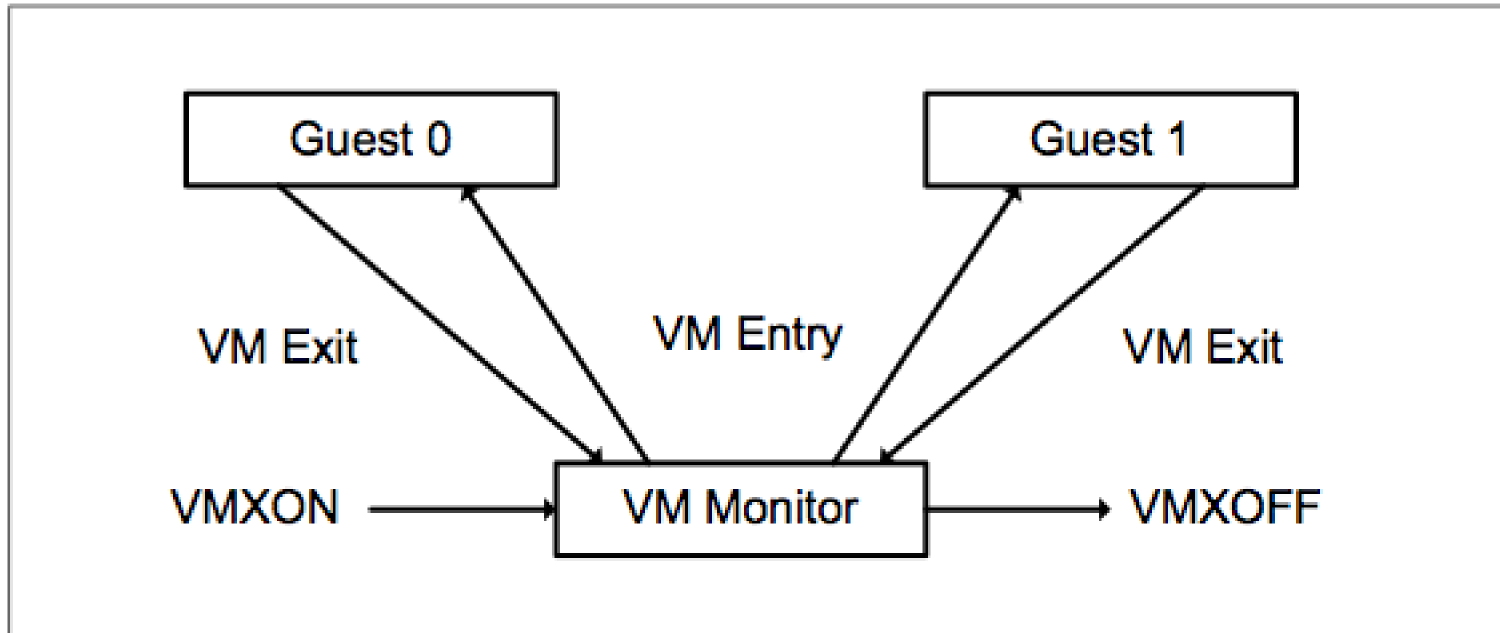


Figure 23-1. Interaction of a Virtual-Machine Monitor and Guests

VMCLEAR

VMCLEAR—Clear Virtual-Machine Control Structure

Opcode	Instruction	Description
66 0F C7 /6	VMCLEAR m64	Copy VMCS data to VMCS region in memory.

Description

This instruction applies to the VMCS whose VMCS region resides at the physical address contained in the instruction operand. The instruction ensures that VMCS data for that VMCS (some of these data may be currently maintained on the processor) are copied to the VMCS region in memory. It also initializes parts of the VMCS region (for example, it sets the launch state of that VMCS to clear). See Chapter 21, "Virtual-Machine Control Structures," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

The operand of this instruction is always 64 bits and is always in memory. If the operand is the current-VMCS pointer, then that pointer is made invalid (set to FFFFFFFF_FFFFFFFFH).

Note that the VMCLEAR instruction might not explicitly write any VMCS data to memory; the data may be already resident in memory before the VMCLEAR is executed.

VMPTRLD

VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /6	VMPTRLD m64	Loads the current VMCS pointer from memory.

Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.[1]

The operand of this instruction is always 64 bits and is always in memory.

VMControlStructure

- Offset 0: VMCS revision ID
  - Same as VMXON region
- Offset 4: VMX abort indicator
- Offset 8: VMCS data (later)
- Size determined by IA32_VMX_BASIC MSR
  - Same as VMXON region
- Once allocated, not to be directly accessed (except for putting the revision ID)
- Instead, use VMREAD/VMWRITE with desired field encodings.

VMWRITE

VMWRITE—Write Field to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 79	VMWRITE r64, r/m64	Writes a specified VMCS field (in 64-bit mode)
0F 79	VMWRITE r32, r/m32	Writes a specified VMCS field (outside 64-bit mode)

Description

Writes to a specified field in the VMCS specified by a secondary source operand (register only) using the contents of a primary source operand (register or memory).

VMX Field Encodings

- Every field of the VMCS is encoded by a 32-bit value
- APPENDIX B for complete description
- For 64-bit fields using the “high” access type
  - A VMREAD returns the value of bits 63:32 of the field in bits 31:0 of the destination operand
  - in 64-bit mode, bits 63:32 of the destination operand are zeroed out

Bit position	Contents
0	Access type (0 = full; 1 = high); must be full for 16-bit, 32-bit, and natural- width fields
9:1	Index
11:10	Type: 0: control 1: read-only data 2: guest state 3: host state
12	Reserved to 0
14:13	Width 0: 16-bit 1: 64-bit 2: 32-bit 3: natural width
31:15	Reserved to 0

Sample VMCS Field Encodings in C

```
/* VMCS Encodings */
enum {
   GUEST_ES_SELECTOR = 0x00000800,
   GUEST_CS_SELECTOR = 0x00000802,
   GUEST_SS_SELECTOR = 0x00000804,
   GUEST_DS_SELECTOR = 0x00000806,
   GUEST_FS_SELECTOR = 0x00000808,
   GUEST_GS_SELECTOR = 0x0000080a,
   GUEST_LDTR_SELECTOR = 0x0000080c,
   GUEST_TR_SELECTOR = 0x0000080e,
   HOST_ES_SELECTOR = 0x00000c00,
   HOST_CS_SELECTOR = 0x00000c02,
   HOST_SS_SELECTOR = 0x00000c04,
   HOST_DS_SELECTOR = 0x00000c06,
   HOST_FS_SELECTOR = 0x00000c08,
   HOST_GS_SELECTOR = 0x00000c0a,
   HOST_TR_SELECTOR = 0x00000c0c,
   IO_BITMAP_A = 0x00002000,
   IO_BITMAP_A_HIGH = 0x00002001,
   IO_BITMAP_B = 0x00002002,
  IO_BITMAP_B_HIGH = 0x00002003,
   MSR_BITMAP = 0x00002004,
   MSR_BITMAP_HIGH = 0x00002005,
   VM_EXIT_MSR_STORE_ADDR = 0x00002006,
   VM_EXIT_MSR_STORE_ADDR_HIGH = 0x00002007,
   VM_EXIT_MSR_LOAD_ADDR = 0x00002008,
   VM_EXIT_MSR_LOAD_ADDR_HIGH = 0x00002009,
...
};
```

Example VMWRITE

- VMREAD/VMWRITE operate on the *current* VMCS
  - Only one *current* VMCS at a time

```
# set guest RIP to 0x12345678 (pure assembly)
mov rax, 0681Eh    /* GUEST_RIP VMCS field*/
mov rbx, 12345678h
vmwrite rax, rbx
```

VMLAUNCH/VMRESUME

VMLAUNCH/VMRESUME—Launch/Resume Virtual Machine

Opcode	Instruction	Description
0F 01 C2	VMLAUNCH	Launch virtual machine managed by current VMCS.
0F 01 C3	VMRESUME	Resume virtual machine managed by current VMCS.

Description

Effects a VM entry managed by the current VMCS.

- The VMLAUNCH instruction requires a VMCS whose launch state is “clear”
- The VMRESUME instruction requires a VMCS whose launch state is “launched”

VM “Launch” State

- The launch state of a VMCS determines which VM-entry instruction should be used with that VMCS: the VMLAUNCH instruction requires a VMCS whose launch state is “clear”; the VMRESUME instruction requires a VMCS whose launch state is “launched”.
- In other words, if the launch state of the current VMCS is “clear,” VMLAUNCH changes the launch state to “launched.”

VM “Launch” State (2)

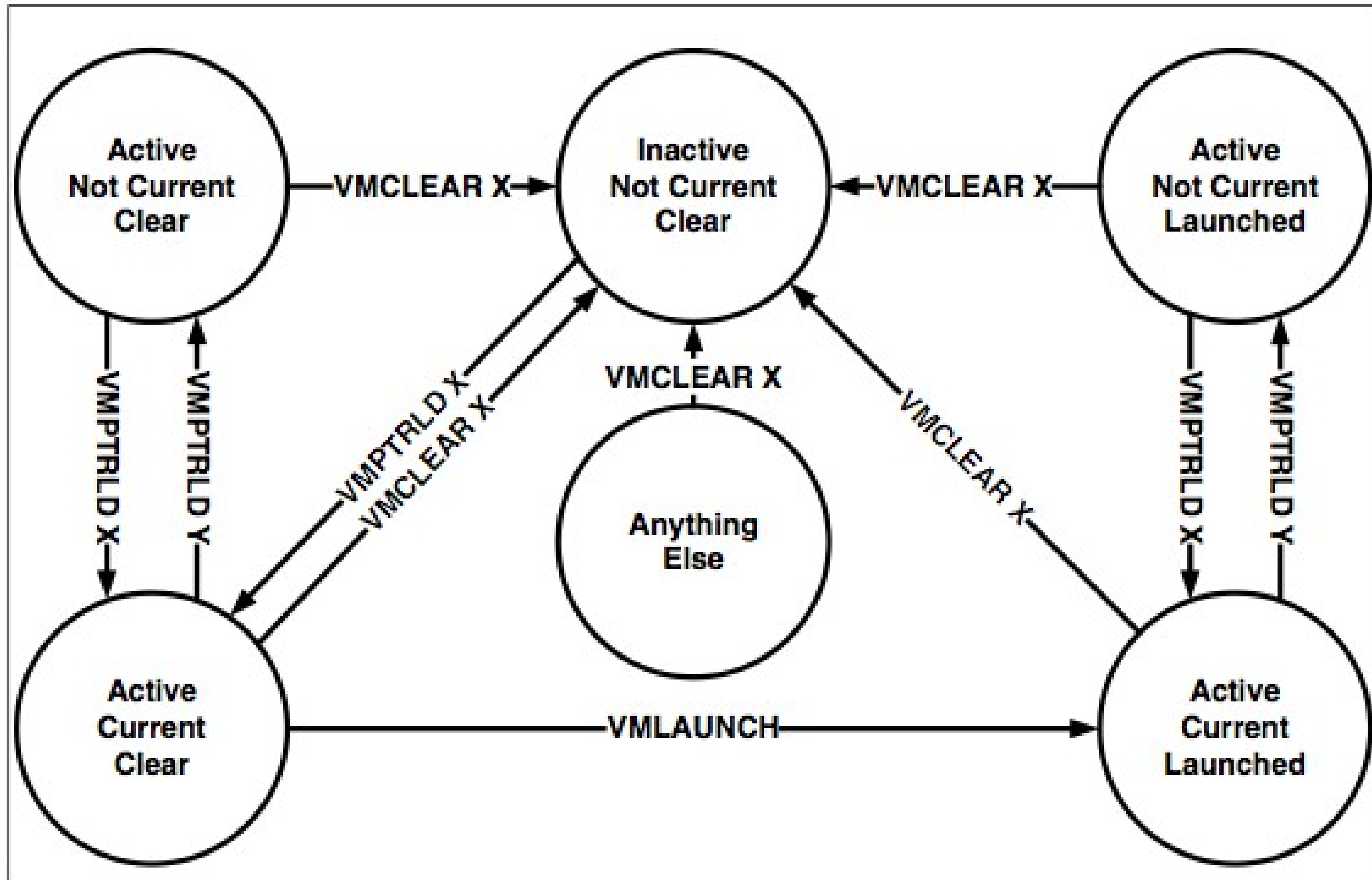


Figure 24-1. States of VMCS X

VMREAD

VMREAD—Read Field from Virtual-Machine Control Structure

Opcode	Instruction	Description
0F 78	VMREAD r/m64, r64	Reads a specified VMCS field (in 64-bit mode).
0F 78	VMREAD r/m32, r32	Reads a specified VMCS field (outside 64-bit mode).

Description

Reads a specified field from the VMCS and stores it into a specified destination operand (register or memory).

VMPTRST

VMPTRST—Store Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /7	VMPTRST m64	Stores the current VMCS pointer into memory.

Description

Stores the current-VMCS pointer into a specified memory address. The operand of this instruction is always 64 bits and is always in memory.

VM Read/Write in our VMM

In our amd64/amd64.asm:

```
_ReadVMCS PROC
    vmread rdx, rcx
    mov rax, rdx
    ret
_ReadVMCS ENDP

_WriteVMCS PROC
    vmwrite rcx, rdx
    ret
_WriteVMCS ENDP
```

And Back in C

- Our prototype functions for binding assembly functions
- ULONG64 _ReadVMCS(ULONG32 Encoding);
- VOID _WriteVMCS(ULONG32 Encoding, ULONG64 Value);

Lab: Introducing Virtdbg

- Purpose
  - Familiarize yourself with the code base
  - Prepare to make some edits to the code
- Steps
  - Read control flow RTF document
  - Navigate the functions in Visual Studio
  - Optional: Fill in any missing control flow notes in the RTF document

VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. Host state
  3. Execution control fields
  4. Exit control fields
  5. Entry control
  6. VM exit info

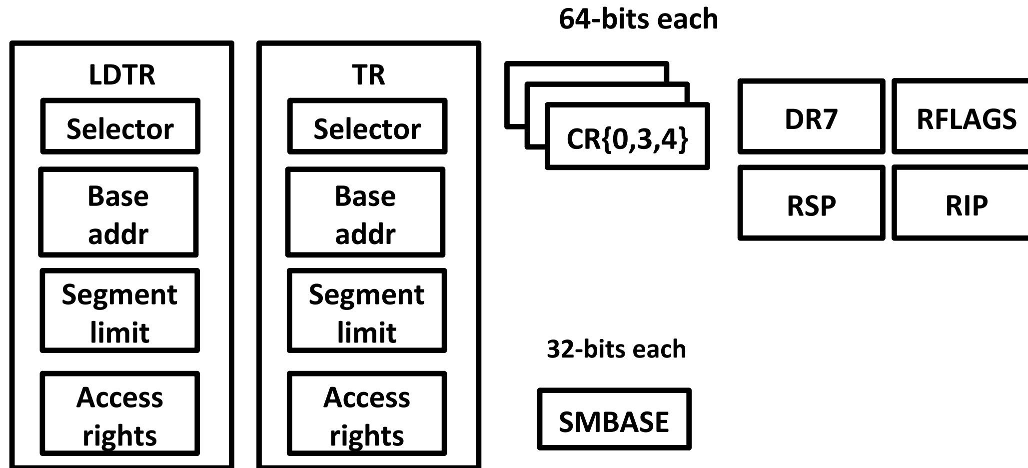
VMCS Data Organization

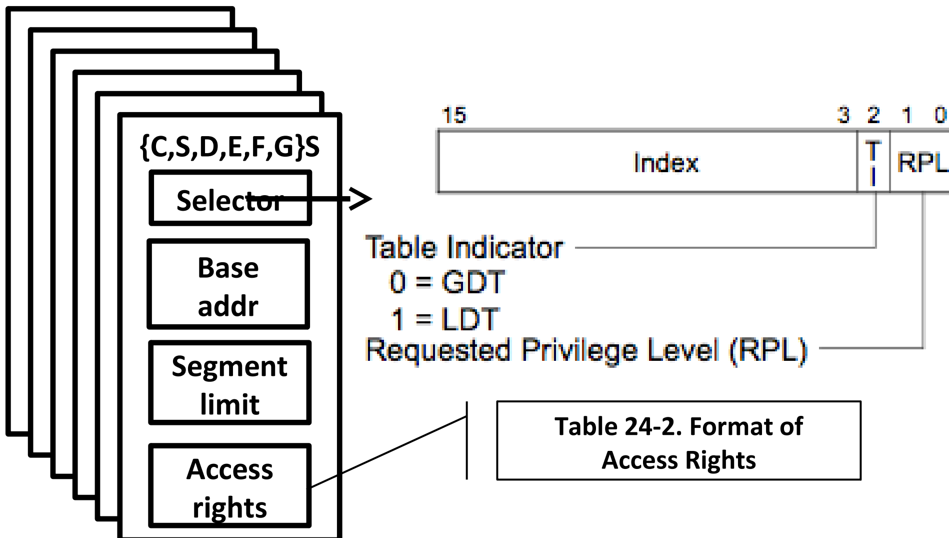
- Organized into 6 categories
  1. **Guest state**
  2. Host state
  3. Execution control fields
  4. Exit control fields
  5. Entry control
  6. VM exit info

VMCS Guest State

- Describes the values of the registers the CPU will have after next VMEntry
- Divided into *Register* and *Non-register* state
- Use of certain fields depends on the “1-setting” of various VM controls
  - E.g., 1-setting of EPT control →PDPTE0 – PDPTE3
- VMCS link pointer is **unused** and **reserved** to FFFFFFFF_FFFFFFFFh

VMCS: Guest Register State





IA32_DEBUGCTL

IA32_SYSENTER_CS (32b)

IA32_SYSENTER_ESP

IA32_SYSENTER_EIP

IA32_PERF_GLOBAL_CTRL*

IA32_PAT*

IA32_EFER*

* - based on 1-setting of respective VM control

SMBASE

- Contains the base address of the logical processor's SMRAM image
- SMM mode related
  - Out of scope of this class
- Should be able to basically ignore it as far as initialization

VMCS Guest Segments

- Access rights come from the segment descriptor. Base addr + Segment Limit + Access rights form the “descriptor cache” of each segment register.
- These data are included in the VMCS because it is possible for a segment register’s descriptor cache to be inconsistent with the segment descriptor in memory (in the GDT or the LDT) referenced by the segment register’s selector.

Grabbing/Setting the Guest segments

- In practice for our toy VMM we will grab the host's existing values and insert them into our VMCS data structure with VMWRITEs
- We will call some functions (written in our assembly file) from our C code to grab the current values
  - i.e. CR0, CR3, CR4 and {C,S,D,E,F,G}S

Segment access rights format

Bits	Field
3:0	Segment type
4	S – Descriptor type (0=system, 1=code or data)
6:5	DPL – descriptor priv. level
7	P – Segment present
11:8	Reserved
12	AVL – Available for use by system software
13	Reserved (except for CS) L – 64-bit mode active (for CS only)
14	D/B – Default operation size (0=16-bit segment, 1=32-bit segment)
15	G – Granularity
16	Segment unusable (0=usable, 1=unusable)
31:17	Reserved

Once again in C … with feeling!

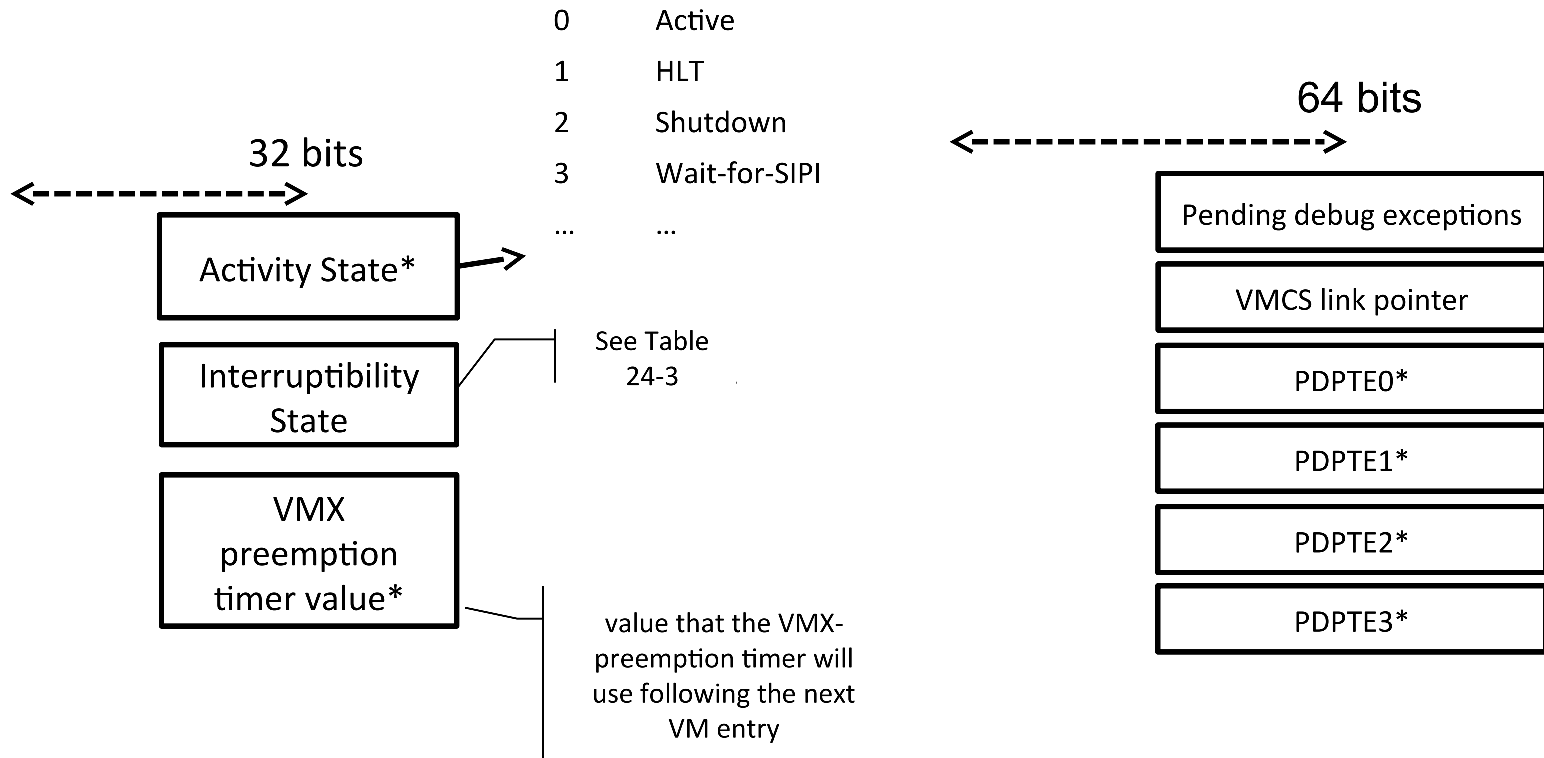
```
typedef union
{
    USHORT UCHARs;
    struct
    {
        USHORT type:4;      /* 0;  Bit 40-43 */
        USHORT s:1;         /* 4;  Bit 44 */
        USHORT dpl:2;       /* 5;  Bit 45-46 */
        USHORT p:1;         /* 7;  Bit 47 */
        // gap!   (this will be explained later)
        USHORT avl:1;       /* 8;  Bit 52 */
        USHORT l:1;         /* 9;  Bit 53 */
        USHORT db:1;        /* 10; Bit 54 */
        USHORT g:1;         /* 11; Bit 55 */
        USHORT Gap:4;
    }   fields;
} SEGMENT_ATTRIBUTES;
```

This is a copy of bit 40:47 & 52:55 of the segment descriptor

Lab: Guest segments in Virtdbg

- Locate in virtdbg the guest segment initialization code
- Create high-level write-up on what you think is happening
- Everyone compare notes
  - Learn from each other

VMCS: Guest Non-register State



* - based on 1-setting of respective VM control

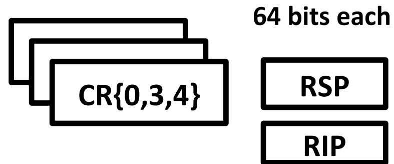
VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. **Host state**
  3. Execution control fields
  4. Exit control fields
  5. Entry control
  6. VM exit info

VMCS Host State Area

- Tells the CPU how to return to the VMM after a VMExit
- Consists of register states and MSRs
  - Again some of the MSRs depend on “1-setting” of VM controls

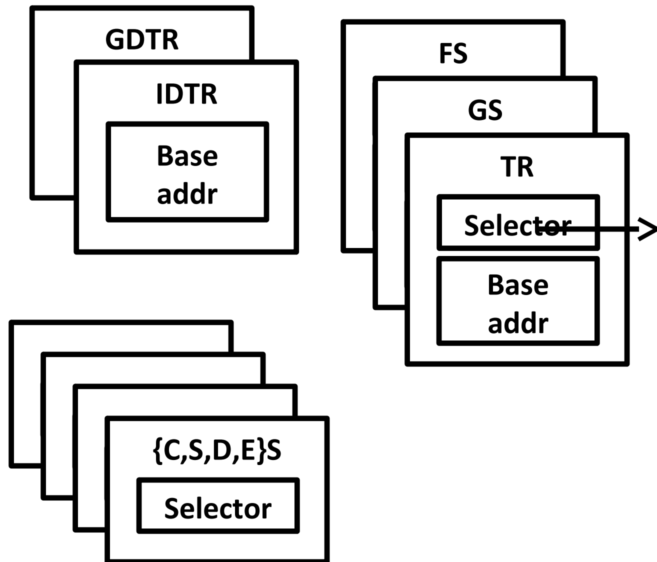
VMCS: Host State Area

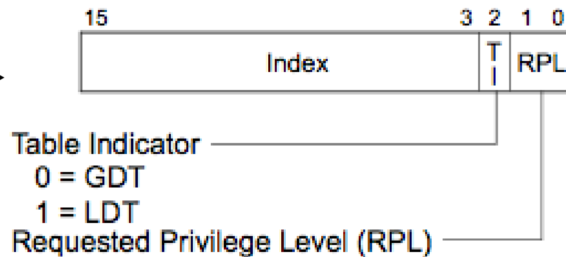


64 bits each

RSP

RIP





IA32_SYSENTER_CS (32)

IA32_SYSENTER_ESP

IA32_SYSENTER_EIP

IA32_PERF_GLOBAL_CTRL*

IA32_PAT*

IA32_EFER*

* - based on 1-setting of respective VM control

VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. Host state
  3. **Execution control fields**
  4. Exit control fields
  5. Entry control
  6. VM exit info

Reserved controls/default settings

- Certain VMX controls are reserved and must be set to a specific value
  - The specific value to which a reserved control must be set is its **default setting**
- Discover the default setting of a reserved control by consulting the appropriate VMX capability MSR
- Partitioned into 3 sets
  - Always-flexible, Default0, Default1
- **Bit 55** of the IA32_VMX_BASIC MSR is used to indicate whether any of the default1 controls may be 0

Memory virtualization

- Volume 3 Section 31.3 in the manual
- Can get complex to implement.
- Choices
  - Brute Force
  - Extended Page Table (EPT) + VPID = Virtual TLB

Memory Virtualization: Brute Force

- Intercept all CR3 load/store
- Use shadow page tables
  - You tell me… what do you think a shadow page table is?
- Ensure all guest manipulation of guest-page hierarchy remains consistent
- Poor performance, doesn't leverage hardware mechanisms (EPT/VPID)

Memory Virtualization: Virtual TLB

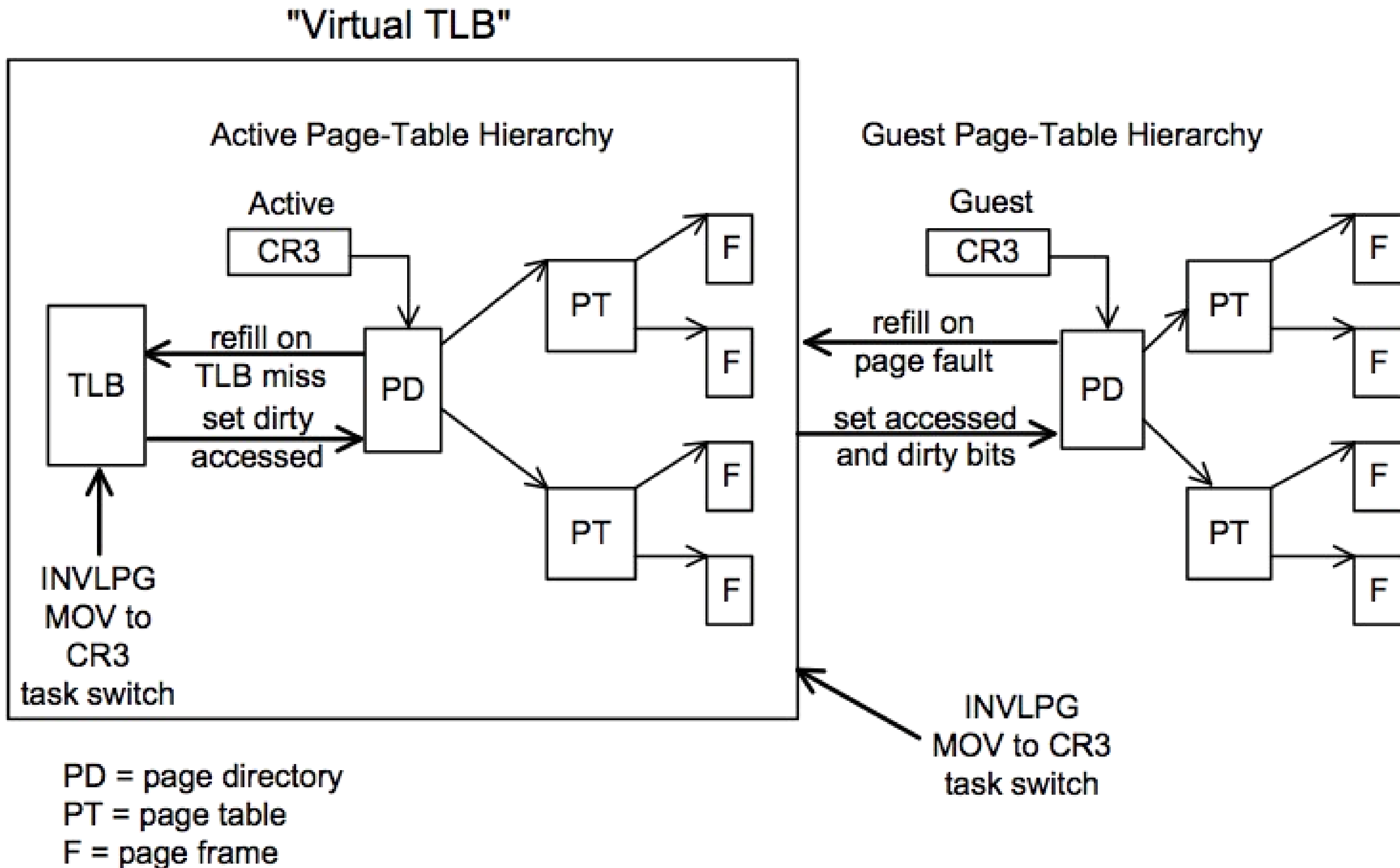


Figure 31-1. Virtual TLB Scheme

VMCS: Execution Control

- Controls what is not allowed to execute in the VM (VMX non-root) without a VMExit, i.e.:
  - Load/store MSRs
  - I/O access in/out
  - Load/store of CR3
  - Read shadows for CR0/CR4
  - Interrupts (i.e., INT3)
  - {RDTSC, TSC MSR} offsetting
  - More…

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

Execution controls not discussed

- APIC access
  - Plays a role in power management
  - Allow guest to be alerted to AC->battery?
- Executive VMCS pointer
  - Plays a role with SMM
  - We don't know enough about SMM, save for another class.
- PAUSE-loop exiting
  - Deals with guest using PAUSE instruction in a loop

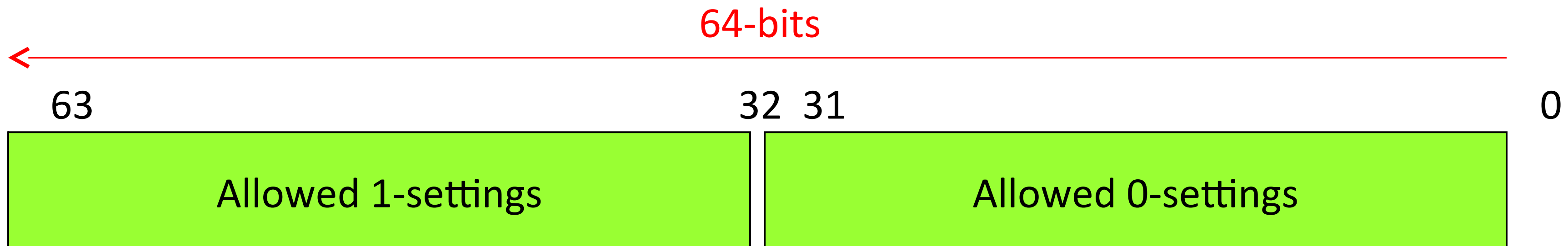
VMCS: Execution Control Fields

- **Pin-based (asynchronous) controls**
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

Pin-based Execution Controls

- One 32-bit value that controls **asynchronous** events in VMX non-root
  - External-interrupt exiting (bit 0)
  - Non maskable interrupt (NMI) exiting (bit 3)
  - Virtual NMIs (bit 5)
  - VMX preemption timer (bit 6)
- Supported settings governed by IA32_VMX_PINBASED_CTLS MSR

MSR: IA32_VMX_PINBASED_CTLS (index 0x481)



- Reports on the allowed settings of **most** of the pin-based VM-execution controls
- **Bits 31:0** indicate the **allowed 0-settings** of these controls.
  - VM entry **allows** control X (i.e. bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0;
  - If bit X in the MSR is set to 1, VM entry fails if control X is 0.
- **Bits 63:32** indicate the **allowed 1-settings** of these controls.
  - VM entry **allows** control X to be 1 if bit 32+X in the MSR is set to 1;
  - If bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

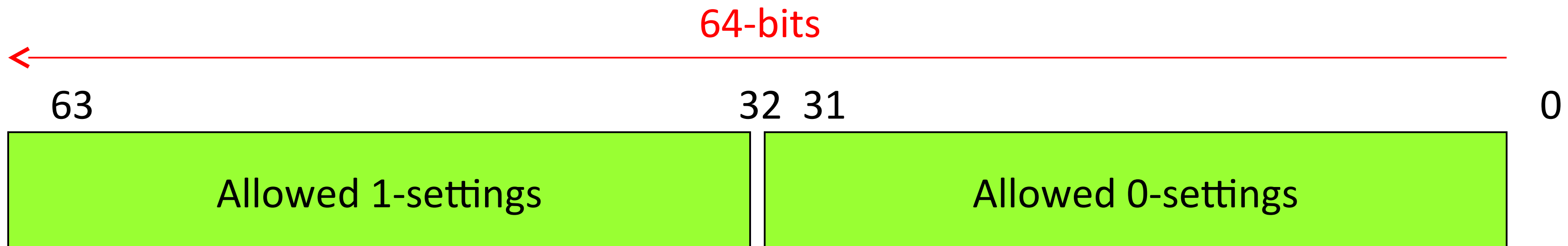
VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- **Processor-based (synchronous) controls**
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

Processor-based Execution Controls

- Two 32-bit values
  - Primary and Secondary controls
- Controls handling of synchronous events
  - i.e., events caused by execution of specific instructions
- See Table 24-6, 24-7 for full description (partial description follows)

MSR: IA32_VMX_PROCBASED_CTLS (index 0x482)



- Reports on the allowed (based on h/w support) settings of **most** of the primary processor-based VM-execution controls
- **Bits 31:0** indicate the **allowed 0-settings** of these controls.
  - VM entry **allows** control X (i.e. bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0;
  - If bit X in the MSR is set to 1, VM entry fails if control X is 0.
- **Bits 63:32** indicate the **allowed 1-settings** of these controls.
  - VM entry **allows** control X to be 1 if bit 32+X in the MSR is set to 1;
  - If bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

Processor-based Execution Controls

- Primary (partial description)
  - 1-setting of ‘use time stamp counter offsetting’ (bit 3)
  - RDTSC exiting control (bit 12)
  - CR3 load exiting (bit 15), CR3 store (bit 16)
  - Activate secondary controls (bit 31)
  - 1-setting of ‘use I/O bitmaps’ (bit 25)
  - 1-setting of ‘use MSR bitmaps’ (bit 28)
  - Activate secondary controls (bit 31)
- Secondary (partial description)
  - Enable EPT (bit 1)
  - Enable VPID (bit 5)
  - Enable VM functions (bit 13)

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- **Exception bitmap**
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

Exception Bitmap

- 32-bit field
  - One bit for each exception
- When an exception occurs (in guest)
  - If the bit is 0 in bitmask
  - Exception delivered through guest IDT normally
  - Otherwise
  - Causes VMExit
- E.g., INT3 exiting
  - Set bit 3 in Exception Bitmap

Exception Bitmap: Page faults

- Special case for page faults (vector 14)
- If software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the **page-fault error-code mask** and **match fields** each to 0x00000000.
- If software desires VM exits on **no page faults**, it can set bit 14 in the exception bitmap to 1, the **page-fault error-code mask field** to 0x00000000, and the **page-fault error-code match field** to 0xFFFFFFFF.

page-fault error-code mask/match fields

- VMRead/VMWrite VMCS field 0x4006 to get Page-fault error-code mask value
- VMRead/VMWrite VMCS field 0x4008 to get Page-fault error-code match value

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- **I/O bitmap addresses**
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

I/O Bitmap Addresses

- Two 4K bitmaps (**A** and **B**)
- **A** contains one bit for each I/O port in range 0000h through 7FFFh
- **B** contains one bit for each I/O port in range 8000h through FFFFh
- Used only with "1-setting" of "use I/O bitmaps" control
  - If bitmaps corresponding to a port is 1, execution of an I/O instruction causes a VM exit

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- **Timestamp Counter offset**
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps

Time-Stamp Counter Offset

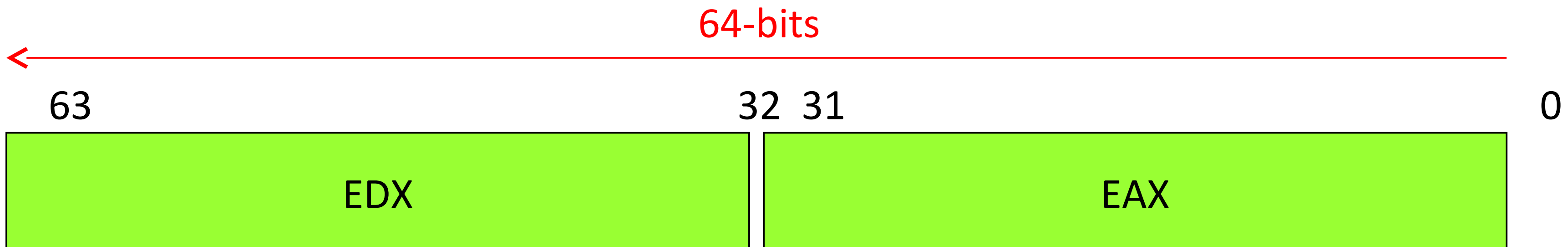
- Controlled by "1-setting" of "use TSC offsetting" option in Primary Processor base control (bit 3)
- Additionally controls RDMSR of IA32_TIME_STAMP_COUNTER
- Signed addition of signed TSC offset and TSC
- Possibly eliminates need for 1-setting of RDTSC-exiting:
  - potential performance optimization
- TSC Offsetting is page 1337 of my Intel manual. Yeah, it's pretty leet.

The Time-Stamp Counter

- 64-bit MSR first introduced in the Pentium
- It increments once every CPU clock-cycle, starting from 0 when power is turned on
- "It won't overflow for at least ten years" (2006*)
- Unprivileged programs (ring3) *normally* can access, it via the RDTSC instruction

* http://cs.usfca.edu/~cruse/cs630f06/lesson27.ppt

Using the TSC




```
time0      dq    0            ; saves starting value from the TSC
time1      dq    0            ; saves concluding value from TSC

     ; how you can measure CPU clock-cycles in a code-fragment
     rdtsc                  ; read the Time-Stamp Counter
     mov  time0+0, eax      ; save least-significant longword
     mov  time0+4, edx      ; save most-significant longword
     ; <Your code-fragment to be measured goes here>
     rdtsc                  ; read the Time-Stamp Counter
     movl time1+0, eax      ; save least-significant longword
     movl time1+4, edx      ; save most-significant longword
     ; now subtract starting-value 'time0' from ending value 'time1'
```

Modified to masm from: http://cs.usfca.edu/~cruse/cs630f06/lesson27.ppt

The TSC as an MSR

- The Time-Stamp Counter is MSR number 0x10
- To write a new 64-bit value into the TSC, you load the desired 64-bit value into the EDX:EAX register-pair, you put the MSR ID-number 0x10 into register ECX, then you execute **wrmsr**
  - To modify a guest TSC use the appropriate VMCS field with VMWRITE

Lab: cpuid hooking

- Purpose:
  - Show that specific VM exit conditions are observed by the VMM
  - Hook cpuid and return our own CPU string
- Steps
  - Modify virtdbg to hook cpuid leaf 0 (eax=0) and return something other than GenuineIntel in appropriate registers
  - Run CPUID either in ring 0 or ring 3 to get the cpuid eax=0 brand string

- sc stop virtdbg

Lab: TSC Offsetting

- Modify Virtdbg to perform TSC offsetting
  - Check your work by running RDTSC in userspace or kernel space
  - Does it work as you expected?

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- **CR0/CR4 guest/host masks**
- CR3 targets
- MSR Bitmaps

Read Shadows

- Reads of CR0 and CR4 don't cause exits.
- Instead, these use "shadows" configured by the VMM in the respective guest's VMCS with the guest-expected values.
  - Why? (poll class)

CR0/CR4 Refresher

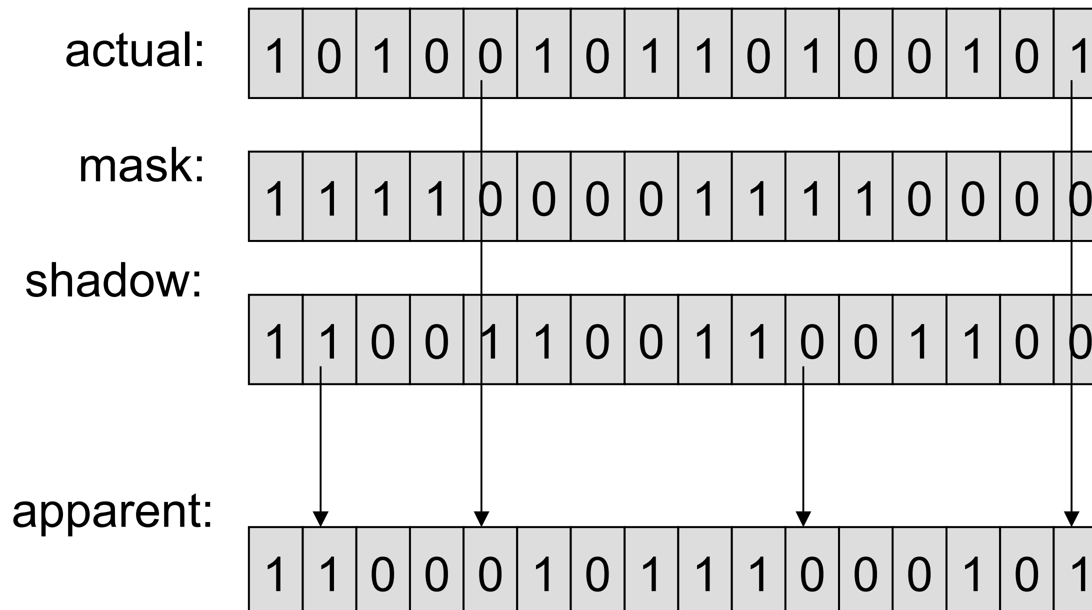
- CR0 (64 bits)
  - Controls things like **paging**, **memory cache** (and write-back), page **write protection**, **protected mode**
- CR4 (64 bits)
  - Controls things like Virtual-8086 mode, enabling SSE, enabling performance counters, PAE paging, page size

Masks and Shadows for CR0/CR4

- Controls execution of instructions that read or modify CR4/CR0
  - i.e., CLTS, LMSW, MOV to/from CR, and SMSW
- Host/guest mask determines who “owns” that bit (guest or host) in CR0/CR4
- For bits set to 1 in the mask, these are owned by host
  - Guest bit-**setting** events
  - Bits set in the mask that differ from respective shadow value will cause VMExit
  - Guest bit-**read** event for bit in bitmask will read from corresponding shadow register
- For bits set to 0 in the mask, these are owned by guest
  - Load/store are unhindered

Cloak and Dagger

- Where a bit is masked, the shadow bit appears
- Where a bit is not masked, the actual bit appears



VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- **CR3 targets**
- MSR Bitmaps

Memory Virtualization (1)

- Provides guest software with contiguous “guest physical” address space starting at zero and extending to the maximum address supported by the guest virtual processor’s physical address width
- The VMM utilizes guest-physical-to-host-physical address mapping to locate all or portions of the guest physical address space in host memory
- The VMM requires the guest-to-host memory mapping to be at page granularity

Memory Virtualization (2)

- Memory virtualization is accomplished using a combination of setting VM exit conditions on specific instructions (i.e. mov to/from cr3) and/or with an Extended Page Table (EPT) mechanism possibly assisted by a translation caching mechanism called Virtual Processor Identifier (VPID)
- For Bluepill we don't need to use some of these mechanisms… though it may make sense to employ some mechanisms to protect the VMM memory from the guest
- More details on memory virtualization will be discussed later. (Part 3)

CR3 Target Controls

- CR3-target values
  - Allows for an exception to the rule of exiting for all MOV to/from CR3. Obviously for performance.
  - Does not cause a VM exit if its source operand matches one of these values.
- IA32_VMX_MISC bits 24:16 indicate the # of CR3-target values supported by the processor
  - In practice most processors only support **4** targets
  - Intel manual is confusing here because they assume 4, but recommend using the IA32_VMX_MISC MSR later.

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- **MSR Bitmaps**

MSR Bitmaps

- Processor must support 1-setting of the "use MSR bitmaps" VM-execution control
- Partitioned into four 1KB contiguous blocks
  1. Read bitmap for low MSRs
  2. Read bitmap for high MSRs
  3. Write bitmap for low MSRs
  4. Write bitmap for high MSRs
- If the bitmaps are used, an execution of RDMSR or WRMSR causes a VM exit if the value of RCX is in neither of the ranges covered by the bitmaps
- Or if the appropriate bit in the MSR bitmaps (corresponding to the instruction and the RCX value) is 1
- VMCS field 0x2004 stores the base address of the MSR bitmaps

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps
- **EPTP**

Extended Page Table Pointer (EPTP)

- Contains the base address of the EPT PML4 table
- VMCS field encoding 0x201A

Table 24-8. Format of Extended-Page-Table Pointer

Bit Position(s)	Field
2:0	EPT paging-structure memory type (see Section 28.2.4): 0 = Uncacheable (UC) 6 = Write-back (WB) Other values are reserved.[1]
5:3	This value is 1 less than the EPT page-walk length (see Section 28.2.2)
11:6	Reserved
N-1:12	Bits N-1:12 of the physical address of the 4-KByte aligned EPT PML4 table[2]
63:N	Reserved

Do you EPT?

- The EPT paging structures are similar to those used to translate linear addresses for 64-bit paging
- When the "enable EPT" VM-execution control is 1, the identity of **guest-physical addresses** depends on whether paging in the guest is enabled
  - If CR0.PG = 0, each linear address is treated as a guest-physical address.
  - If CR0.PG = 1, guest-physical addresses are those derived from the contents of control register CR3 and the guest paging structures.
- A logical processor uses EPT to translate guest-physical addresses only when those addresses are used to access memory.

EPT Translation (1)

- If CR0.PG = 1, the translation of a linear address to a physical address requires multiple translations of guest-physical addresses using EPT

EPT Translation (2)

- Bits 31:22 of the linear address select an entry in the guest page directory located at the guest-physical address in CR3. The guest-physical address of the guest page-directory entry (PDE) is translated through EPT to determine the guest PDE's physical address.

EPT Translation (3)

- Bits 21:12 of the linear address select an entry in the guest page table located at the guest-physical address in the guest PDE. The guest-physical address of the guest page-table entry (PTE) is translated through EPT to determine the guest PTE’s physical address.

EPT Translation (4)

- Bits 11:0 of the linear address is the offset in the page frame located at the guest-physical address in the guest PTE. The guest-physical address determined by this offset is translated through EPT to determine the physical address to which the original linear address translates.

VMCS: Execution Control Fields

- Pin-based (asynchronous) controls
- Processor-based (synchronous) controls
- Exception bitmap
- I/O bitmap addresses
- Timestamp Counter offset
- CR0/CR4 guest/host masks
- CR3 targets
- MSR Bitmaps
- EPTP
- **VPID**

Virtual Processor Identifier (VPID)

- Used to cache information for multiple linear-address spaces
  - “**translations**, which are mappings from linear page numbers to physical page frames,
  - and **paging-structure caches**, which map the upper bits of a linear page number to information from the paging-structure entries used to translate linear addresses…”

Ref: 28.3.1 Information That May Be Cached

Translations

- Linear mappings
  - "Each of these is a mapping from a linear page number to the physical page frame to which it translates, along with information about access privileges and memory typing."
- Guest-physical mappings
  - Guest-physical translations
  - Guest-physical paging-structure-cache entries

More VPID

- The current VPID is 0000H in the following situations:
  - Outside VMX operation. (This includes operation in system-management mode under the default treatment of SMIs and SMM with VMX operation; see Section 29.14.)
  - In VMX root operation.
  - In VMX non-root operation when the “enable VPID” VM-execution control is 0.

VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. Host state
  3. Execution control fields
  4. **Exit control fields**
  5. Entry control
  6. VM exit info

VM Exits (preface)

- VM exits have significant overhead
  1. Begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields (details on this later)
  2. Each field in the guest-state area of the VMCS is written with the corresponding component of current processor state.
  3. Save guest MSR values
  4. Load host state
  5. Load host MSRs
- A problem encountered during a VM exit leads to a VMX abort

VMCS: Exit Control Fields

- VM Exits
  - Occur in response to certain instructions and events in VMX non-root operation
  - i.e., what to load and discard in the case of a VM Exit
- Control fields consist of 2 groups
  1. VM Exit controls
  2. VM Exit controls for MSRs

VM Exit Controls

- 32-bit vector
  - Save debug controls (bit 2)
  - Whether DR7/IA32_DEBUGCTL are saved on VM exit
  - Host addr-space size (bit 9)
  - Whether VM exits occur into 64-bit mode host
  - Load IA32_PERF_GLOBAL_CTRL (bit 12)
  - Whether IA32_PERF_GLOBAL_CTRL is loaded on VM exit
  - Save VMX-preemption timer value (bit 22)
  - Whether pre-emption timer is saved on exit

VM Exit Controls

2	Save debug controls	DR7 and IA32_DEBUGCTL_MSR are saved on VM exit?
9	Host addr space size	64-bit or 32-bit on VM exit (in VMX root mode)
12	Load IA32_PERF_GLOBAL_CTR	IA32_PERF_GLOBAL_CTRL MSR saved on VM exit?
15	Ack. Interrupt on exit	affects VM exits due to external interrupts (see Table 24-10)
18	Save IA32_PAT	IA32_PAT MSR saved on VM exit?
19	Load IA32_PAT	IA32_PAT MSR loaded on VM exit?
20	Save IA32_EFER	IA32_EFER MSR saved on VM exit?
21	Load IA32_EFER	IA32_EFER MSR loaded on VM exit?
22	Save VMX-preemption timer value	VMX-preemption timer saved on VM exit?

*All other bits are reserved, some to 0 and some to 1. Consult the VMX capability MSRs **IA32_VMX_EXIT_CTLS** and **IA32_VMX_TRUE_EXIT_CTLS**

VM Exit Controls for MSRs (1)

- Guest and Host will make use of MSRs
  - Business as usual
- And we can register with the VMM that some MSR reads need not perform VM exits
- But assuming there is a VM exit, i.e., in the case that we want to intercept that MSR read event
  - Then there needs to be a mechanism to store the guest's current MSRs (before exit) and re-load the host's MSRs (so that the VMM can do its job)
  - And later when there is a VM entry we will have shelved the guest's MSRs and it will "do the right thing…"

VM Exit Controls for MSRs (2)

- A VMM may specify lists of MSRs to be stored and loaded on VM exits/entry
- Uses the VM-exit MSR-store-address and the VM-exit MSR-store-count exit control fields
- Each data entry is 16 bytes and expressed by
  - Bits 31:0 store the MSR index
  - Bits 63:32 are reserved (unclear whether to 0 or 1)
  - Bits 127:64 store the MSR data

127 ... 64	63 ... 32	31 ... 0
MSR data	Reserved	MSR index

VM Exit Controls for MSRs

- VM-exit MSR-**store** count (32-bit)
  - Specifies the # of MSRs to be **stored** on VM exit
- VM-exit MSR-**store** address (64-bit)
  - Physical address of the VM-exit *MSR-store area*
- VM-exit MSR-**load** count (32-bit)
  - Contains the # of MSRs to be loaded on VM exit
- VM-exit MSR-**load** address (64-bit)
  - Physical address of the VM-exit *MSR-load area*

VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. Host state
  3. Execution control fields
  4. Exit control fields
  5. **Entry control fields**
  6. VM exit info

VMCS: Entry Control Fields

- 32-bit vector that controls the basic operation of VM entries
- VMCS field 0x4012
- 3 groups
    1. Entry controls
    2. Entry controls for MSRS
    3. Entry controls for Event Injection

Entry controls

2	Load debug controls	DR7 and IA32_DEBUGCTL_MSR are saved on VM exit?
9	IA-32e mode guest	64-bit mode guest? (0=no, 1=yes)
10	Entry to SMM	Put into SMM mode on entry? (0=no, 1=yes)
11	Deactivate dual-monitor treatment	See Section 29.15.7. Set to 0 for VM entry outside SMM (for our purpose)
13	Load IA32_PERF_GLOBAL_CTRL	IA32_PERF_GLOBAL_CTRL MSR is loaded on VM entry?
14	Load IA32_PAT	IA32_PAT MSR <u>loaded</u> on VM entry?
15	Save IA32_EFER	IA32_EFER MSR <u>saved</u> on VM entry?

Entry controls for MSRs

- Similar to exit controls for VM Exit.
  - Except use the VM entry load count and address
  - VM-entry MSR load count: 0x4014 (VMCS field)

VM Event Injection (1)

- VMX operation allows injecting interruptions to a guest virtual machine through the use of VM-entry interrupt-information field in VMCS.
  - Generate event on next VMEnter
  - Happens after all guest state is loaded
- Allows injection of
  - External interrupts
  - Non-maskable interrupts
  - Exceptions (e.g., page faults)
  - Traps

VM Event Injection (2)

- If the interrupt-information field indicates a valid interrupt, exception or trap event upon the next VM entry; the processor will use the information in the field to vector a virtual interruption through the guest IDT after all guest state and MSRs are loaded.

Entry controls for Event Injection

- VM entry can be configured to conclude by delivering an event through the IDT
  - after all guest state and MSRs have been loaded
- Use the VM-entry interruption-information field (VMREAD/VMWRITE 0x4016)

VM Function Controls

VMCS Data Organization

- Organized into 6 categories
  1. Guest state
  2. Host state
  3. Execution control fields
  4. Exit control fields
  5. Entry control
  6. **VM exit info**

VM Exit (1)

- All VM exits load processor state from the host-state area of the VMCS that was the controlling VMCS before the VM exit.
  - This state is checked for consistency while being loaded. Because the host-state is checked on VM entry, these checks will generally succeed.
  - Failure is possible only if host software is incorrect or if VMCS data in the VMCS region in memory has been written by guest software (or by I/O DMA) since the last VM entry.

VM Exit info

- The VM-exit information fields provide details on VM exits due to exceptions and interrupts. This information is provided through the **exit-qualification**, **VM-exit-interruption-information, instruction-length** and **inter-ruption-error-code** fields.

VM Exit info field encodings

```
enum {
...
    VM_EXIT_REASON = 0x00004402,
    VM_EXIT_INTR_INFO = 0x00004404,
VM_EXIT_INTR_ERROR_CODE = 0x00004406,
    VM_EXIT_INSTRUCTION_LEN = 0x0000440c,
    EXIT_QUALIFICATION = 0x00006400,
...
}
```

VMCS: Exit Information fields

- Basic info
  - Exit reason (32-bits)
  - Exit qualification (64-bits)
  - Guest Linear Address (64-bits)
  - Guest Physical Address (64-bits)
- Vectored exit info
- Event delivery exits
- Instruction execution exits
- Error field

VM Exit Error Handling

- Examples
  - There was a failure on storing guest MSRs.
  - There was failure in loading a PDPTR.
  - The controlling VMCS has been corrupted
  - There was a failure on loading host MSRs
  - Machine-check event

VMM Error Handling

- Error conditions may occur during VM entries and VM exits and a few other situations
- Two basic strategies for error handling in VMM
  - Basic error handling: in this approach the guest VM is treated as any other thread of execution. If the error recovery action does not support restarting the thread after handling the error, the guest VM should be terminated.
  - Machine Check Architecture virtualization. In this approach, the VMM virtualizes the MCA events and hardware. This enables the VMM to intercept MCA events and inject an MCA into the guest VM. The guest VM then has the opportunity to attempt error recovery actions, rather than being terminated by the VMM.
- MCA virtualization is out of scope of this class

VMX Errors

Table 5-1. VM-Instruction Error Numbers

Error Number	Description
1	VMCALL executed in VMX root operation
2	VMCLEAR with invalid physical address
3	VMCLEAR with VMXON pointer
4	VMLAUNCH with non-clear VMCS
5	VMRESUME with non-launched VMCS
6	VMRESUME after VMXOFF (VMXOFF and VMXON between VMLAUNCH and VMRESUME)[1]
7	VM entry with invalid control field(s)[2,3]
8	VM entry with invalid host-state field(s)[2]
9	VMPTRLD with invalid physical address
10	VMPTRLD with VMXON pointer
11	VMPTRLD with incorrect VMCS revision identifier
12	VMREAD/VMWRITE from/to unsupported VMCS component
13	VMWRITE to read-only VMCS component
15	VMXON executed in VMX root operation
16	VM entry with invalid executive-VMCS pointer[2]
17	VM entry with non-launched executive VMCS[2]
18	VM entry with executive-VMCS pointer not VMXON pointer (when attempting to deactivate the dual-monitor treatment of SMIs and SMM)[2]
19	VMCALL with non-clear VMCS (when attempting to activate the dual-monitor treatment of SMIs and SMM)
20	VMCALL with invalid VM-exit control fields
22	VMCALL with incorrect MSEG revision identifier (when attempting to activate the dual-monitor treatment of SMIs and SMM)
23	VMXOFF under dual-monitor treatment of SMIs and SMM
24	VMCALL with invalid SMM-monitor features (when attempting to activate the dual-monitor treatment of SMIs and SMM)

Intel Reference
Volume 2B
Section 5.4
Defines 28 error codes
(not all shown here)

VM Entry

```
MOV rax,0681eh       # RIP VMCS identifier
MOV rbx,0            #
VMWRITE rax,rbx      # after next successful VM entry, guest will start with RIP=0
```

Lightning review of Debugging

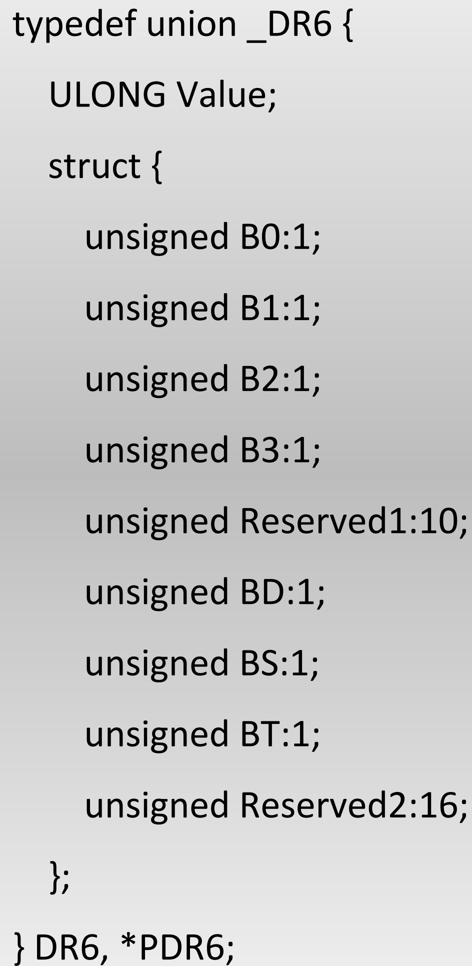
- Registers hold addresses of memory and I/O of BPs/state
  - 8 registers, 4 for BPs (DR0-3), Debug Status (DR6), Debug Status (DR7)
- MSRs monitor branches, interrupts, and exceptions

Debug Registers Review

- Most debuggers also have support for something called a "hardware breakpoint", and these breakpoints are more flexible than software breakpoints in that they can be set to trigger when memory is read or written, not just when it's executed. However only 4 hardware breakpoints can be set.
- There are 8 debug registers DR0-DR7
  - DR0-3 = breakpoint linear address registers
  - DR4-5 = reserved (unused)
  - DR6 = Debug Status Register
  - DR7 = Debug Control Register
- Accessing the registers requires CPL == 0
  - MOV DR, r32
  - MOV r32, DR

Ref: http://opensecuritytraining.info/IntermediateX86.html (Part 4)

DR6 typedef



```c
typedef union _DR6 {
    ULONG Value;
    struct {
        unsigned B0:1;
        unsigned B1:1;
        unsigned B2:1;
        unsigned B3:1;
        unsigned Reserved1:10;
        unsigned BD:1;
        unsigned BS:1;
        unsigned BT:1;
        unsigned Reserved2:16;
    };
} DR6, *PDR6;
```

DR7 typedef

```
typedef union _DR7 {
   ULONG Value;
   struct {
      unsigned L0:1;
      unsigned G0:1;
      unsigned L1:1;
      unsigned G1:1;
      unsigned L2:1;
      unsigned G2:1;
      unsigned L3:1;
      unsigned G3:1;
      unsigned LE:1;
      unsigned GE:1;
      unsigned Reserved1:3;
      unsigned GD:1;
      unsigned Reserved2:2;
      unsigned RW0:2;
      unsigned LEN0:2;
      unsigned RW1:2;
      unsigned LEN1:2;
      unsigned RW2:2;
      unsigned LEN2:2;
      unsigned RW3:2;
      unsigned LEN3:2;
   };
} DR7, *PDR7;
```

Virt + Debugging: play nice? (1)

- Volume 3 section 31.2
- The VMM can program the exception-bitmap to ensure it gets control on debug functions
  - like breakpoint exceptions occurring while executing guest code such as INT3 instructions
- The VMM may utilize the VM-entry event injection facilities described to inject debug or breakpoint exceptions to the guest.

Virt + Debugging: play nice? (2)

- The MOV-DR exiting control bit in the processor-based VM-execution control field can be enabled by the VMM to cause VM exits on explicit guest access of processor debug registers
- Guest software access to debug-related model-specific registers (such as IA32_DEBUGCTL MSR) can be trapped by the VMM through MSR access control features

Virt + Debugging: play nice? (3)

- Debug registers such as DR7 and the IA32_DEBUGCTL MSR may be explicitly modified by the guest (through MOV-DR or WRMSR instructions)
- Or modified implicitly by the processor as part of generating debug exceptions.
- The current values of DR7 and the IA32_DEBUGCTL MSR are saved to guest-state area of VMCS on every VM exit.

Virt + Debugging: play nice? (4)

- DR7 and the IA32-DEBUGCTL MSR are loaded from values in the guest-state area of the VMCS on every VM entry.
- This allows the VMM to properly virtualize debug registers when injecting debug exceptions to guest.
- Similarly, the RFLAGS register is loaded on every VM entry (or pushed to stack if injecting a virtual event) from guest-state area of the VMCS.

Lab: Virtdbg exceptions

End of Section