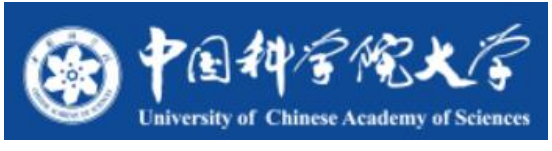
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

处理器虚拟化

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



回顾：RISC-V的特权级

- U: 用户模式，被用于传统应用程序
- S: 管理员模式，被用于操作系统
- H: Hypervisor 模式,支持虚拟机监视器
- M: 机器模式，权限最高，代码是固有可信的，因为它可以在低层次访问机器的实现

级别	编码	名字	缩写
0	00	用户模式	U
1	01	管理员/监管者模式	S
2	10	Hypervisor/保留	H
3	11	机器模式	M

回顾：特权模式实现

- **实现可以提供从特权级 1 到特权级 4 任意级别开始的特权模式，这需要在隔离性和较小的实现代价之间进行折中：**

1. 所有硬件实现必须提供 M-mode
2. 许多 RISC-V 实现还支持至少一个用户模式（U-mode），以对系统的其他部分进行保护，防止被应用程序代码破坏。
3. 管理员模式（S-mode）可被加入，以在管理员级操作系统和 SEE、HAL 之间提供隔离。
4. Hypervisor 模式（H-mode）将在一个虚拟机监视器和 HEE、运行在机器模式的 HAL 之间提供隔离。

级别数量	支持的模式
1	M
2	M+U
3	M+S+U
4	M+H+S+U

回顾：Trap

一个 hart（硬件线程）通常在 U-mode 下运行应用程序，直到某些自陷（例如一个管理员调用或者一个定时器中断）强制切换到一个自陷处理函数（trap handler），这个自陷处理函数通常运行在更特权的模式下。然后这个线程将执行这个自陷处理器函数，它最终在U-mode 下，在引起自陷的指令处或之后，继续线程执行。提升特权基级别的自陷称为垂直自陷（vertical trap），而保持在同样特权级别的自陷称为水平自陷（horizontal trap）。RISC-V特权体系结构提供了将自陷灵活地路由到不同的特权层。

回顾：自陷重定向指令

简单的实现可以将所有的自陷定向到一个 M-mode 自陷处理函数，即使它们的目标是一个较低的特权模式。自陷重定向指令允许 M-mode 自陷处理函数快速地将控制传输到一个较低特权模式的自陷处理函数：

MRTS 指令（Machine Redirect Trap to Supervisor）将自陷处理从 M-mode 转移到 S-mode。将 pc 设置为管理员的自陷处理函数，（这个自陷处理函数入口）被保存在 stvec 寄存器中。另外，mepc、mcause、mbadaddr 寄存器中的值被分别复制到 sepc、scause、sbadaddr 寄存器中。

MRTH 指令（Machine Redirect Trap to Hypervisor）类似，但是是将自陷处理转移到 H-mode 的 htvec。mepc、mcause、mbadaddr 寄存器被分别复制到 hepc、hcause、hbadaddr 寄存器中。

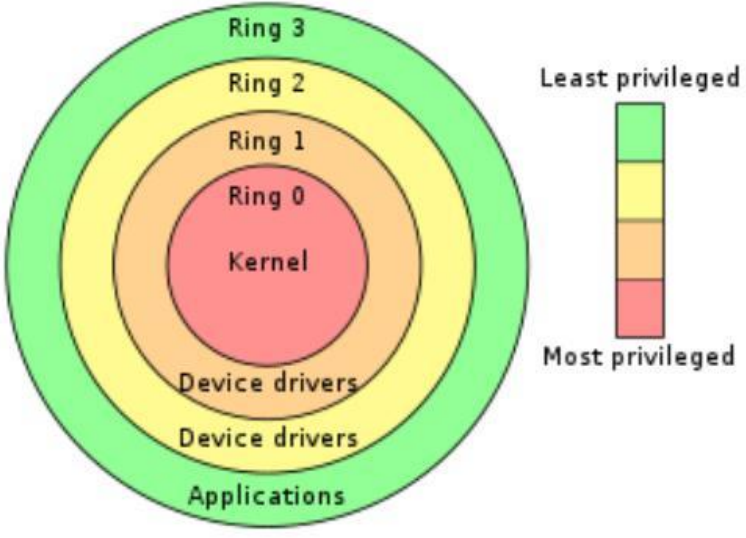
31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRTS	0	PRIV	0	SYSTEM
MRTH	0	PRIV	0	SYSTEM

自陷重定向指令

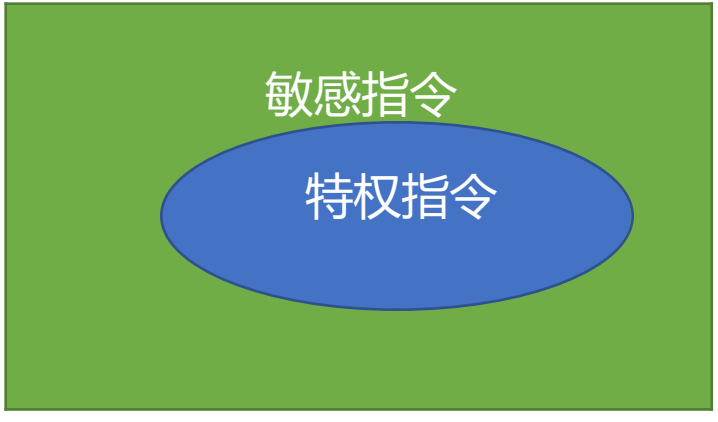
001100000101	00000	000	00000	1110011	MRTS
001100000110	00000	000	00000	1110011	MRTH
001000000101	00000	000	00000	1110011	HRTS

特权指令和敏感指令

- 特权指令：系统资源的分配与管理
  - 改变系统的工作模式
  - 检查用户的访问权限
  - 修改虚拟存储器管理的段表/页表
- 特权级别



- 敏感指令
  - 访问或修改虚拟机模式、机器状态
  - I/O操作



RISC-V是严格的可虚拟化架构

- **特权级别设计：** RISC-V ISA 定义了清晰的特权级别，包括机器模式（Machine Mode）、监管者模式（Supervisor Mode）、用户模式（User Mode）等。这些模式在设计上允许严格的隔离，使得虚拟机监视器（VMM）可以以严格的方式控制和管理不同特权级别的访问，而不需要修改敏感状态或指令。
- **指令集扩展支持：** RISC-V 架构在其规范中包含了用于虚拟化的指令集扩展，如 RISC-V 的虚拟化扩展（RV32H 和 RV64H）。这些扩展提供了对虚拟化相关操作的支持，使得虚拟化软件可以更高效地操作和管理虚拟化环境。

问题：ARM不是严格的可虚拟化架构

- **敏感指令**
  - 读写特殊寄存器或更改处理器状态
  - 读写敏感内存：例如访问未映射内存、写入只读内存
  - I/O指令
- **特权指令**
  - 在用户态执行会触发异常，并陷入内核态

ARM不是严格的可虚拟化架构

- **在ARM中：不是所有敏感指令都属于特权指令**
- **例子: CPSID/CPSIE指令**
  - CPSID和CPSIE分别可以关闭和打开中断
  - 内核态执行：PSTATE.{A, I, F} 可以被CPS指令修改
  - 在用户态执行：CPS 被当做NOP指令，不产生任何效果
    - 不是特权指令
- **怎么解决呢?**

如何处理这些不会下陷的敏感指令?

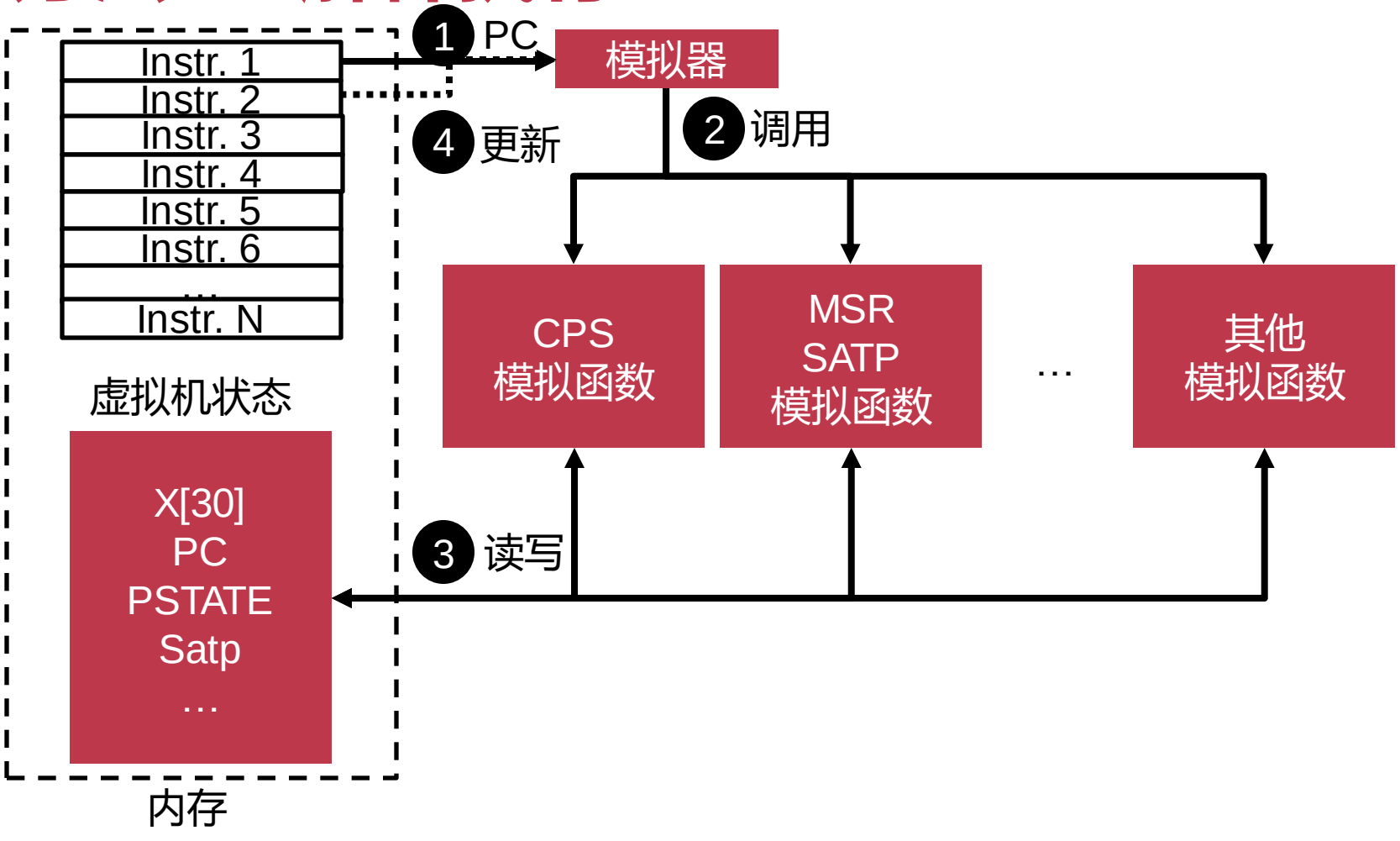
处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态（user mode）

- **方法1：解释执行**
- **方法2：二进制翻译**
- **方法3：半虚拟化**
- **方法4：硬件虚拟化（改硬件）**

方法1：解释执行

- **使用软件方法一条条对虚拟机代码进行模拟**
  - 不区分敏感指令还是其他指令
  - 没有虚拟机指令直接在硬件上执行
- **使用内存维护虚拟机状态**
  - 例如：使用uint64_t x[30]数组保存所有通用寄存器的值

方法1：解释执行


Instr. 1
Instr. 2
Instr. 3
Instr. 4
Instr. 5
Instr. 6
…
Instr. N
1 PC
模拟器
2 调用
4 更新
CPS
模拟函数
MSR
SATP
模拟函数
…
其他
模拟函数
虚拟机状态
X[30]
PC
PSTATE
Satp
…
3 读写
内存

例子：使用解释执行来模拟中断的设置

```
void CPU_Run(void)
{
    while (1) {
        inst = Fetch(CPUState.PC);

        CPUState.PC += 4;

        switch (inst) {
        case ADD:
            CPUState.GPR[rd] = GPR[rn] + GPR[rm];
            break;
        …
        case CLI:
            CPU_CLI(); break;
        case STI:
            CPU_STI(); break;
        }

        if (CPUState.IRQ && CPUState.IE) {
            CPUState.IE = 0;
            CPU_Vector(EXC_INT);
        }
    }
}
```

```
void CPU_CLI(void)
{
    CPUState.IE = 0;
}


void CPU_STI(void)
{
    CPUState.IE = 1;
}


void CPU_Vector(int exc)
{
    CPUState.LR = CPUState.PC;
    CPUState.PC = disTab[exc];
}
```

思考：这种方式有什么优缺点?

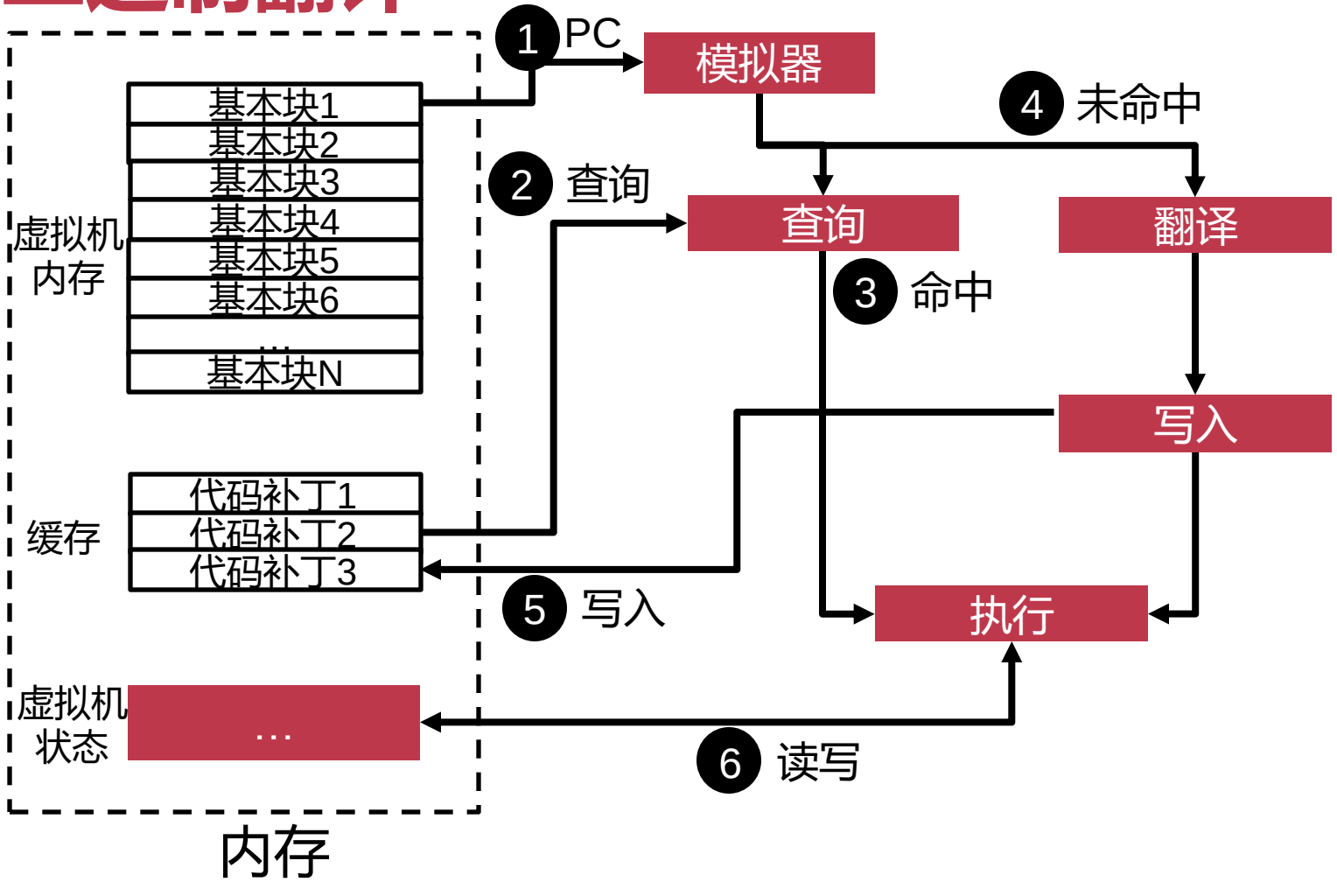
解释执行的优缺点

- **优点：**
  - 解决了敏感函数不下陷的问题
  - 可以模拟不同ISA的虚拟机
  - 易于实现、复杂度低
- **缺点：**
  - 非常慢：任何一条虚拟机指令都会转换成多条模拟指令

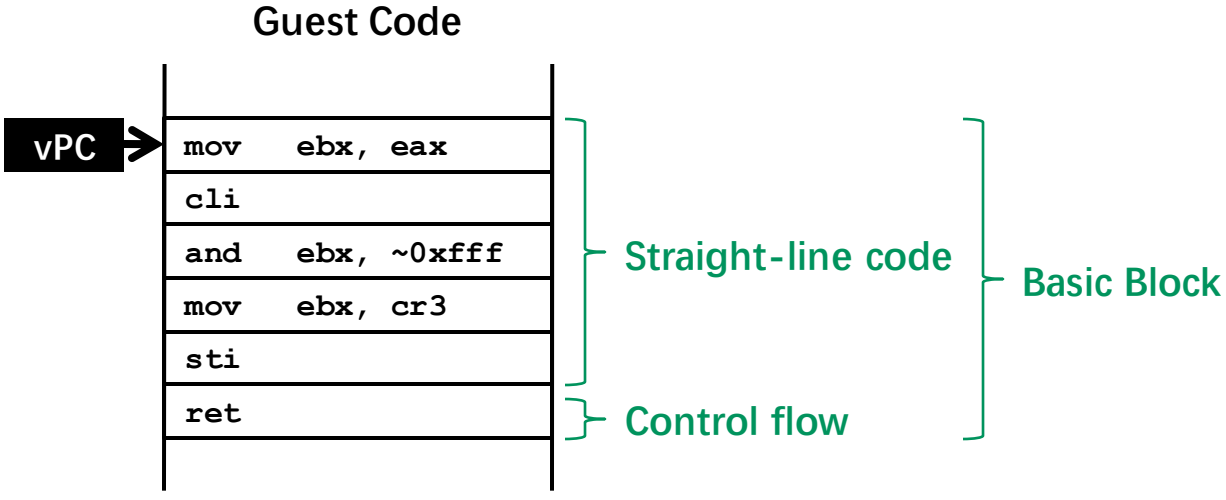
方法2：二进制翻译

- **提出两个加速技术**
  - 在执行前批量翻译虚拟机指令
  - 缓存已翻译完成的指令
- **使用基本块(Basic Block)的翻译粒度（为什么?）**
  - 每一个基本块被翻译完后叫代码补丁

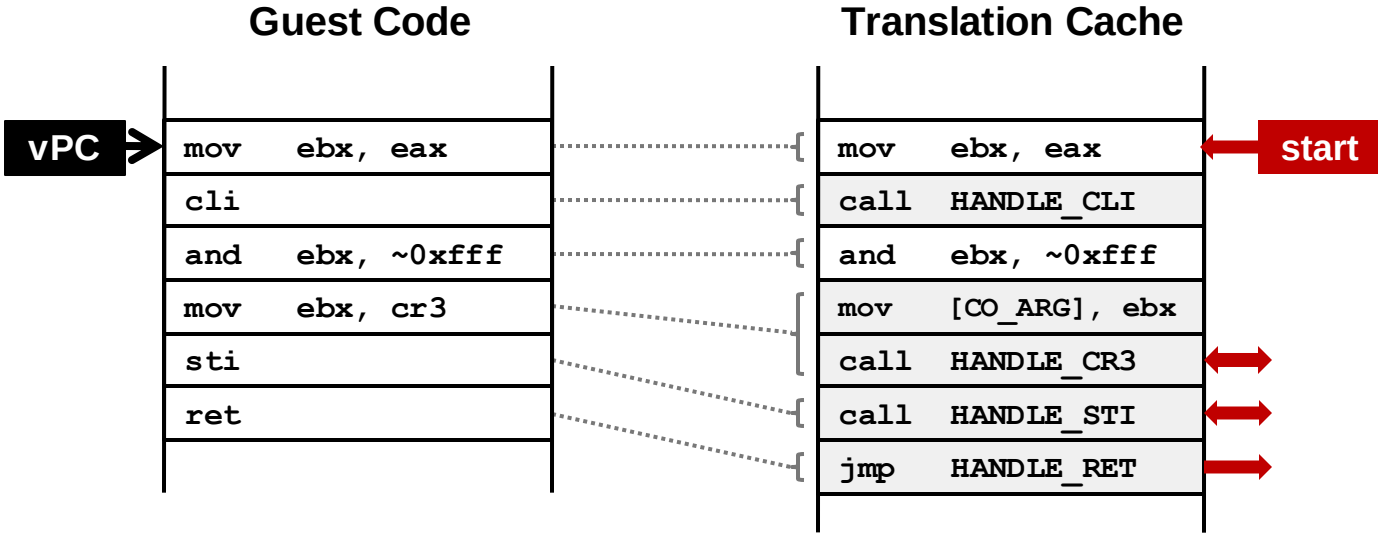
二进制翻译


1 PC
模拟器
4 未命中
2 查询
查询
翻译
3 命中
虚拟机
内存
基本块1
基本块2
基本块3
基本块4
基本块5
基本块6
…
基本块N
写入
缓存
代码补丁1
代码补丁2
代码补丁3
5 写入
执行
虚拟机
状态
…
6 读写
内存

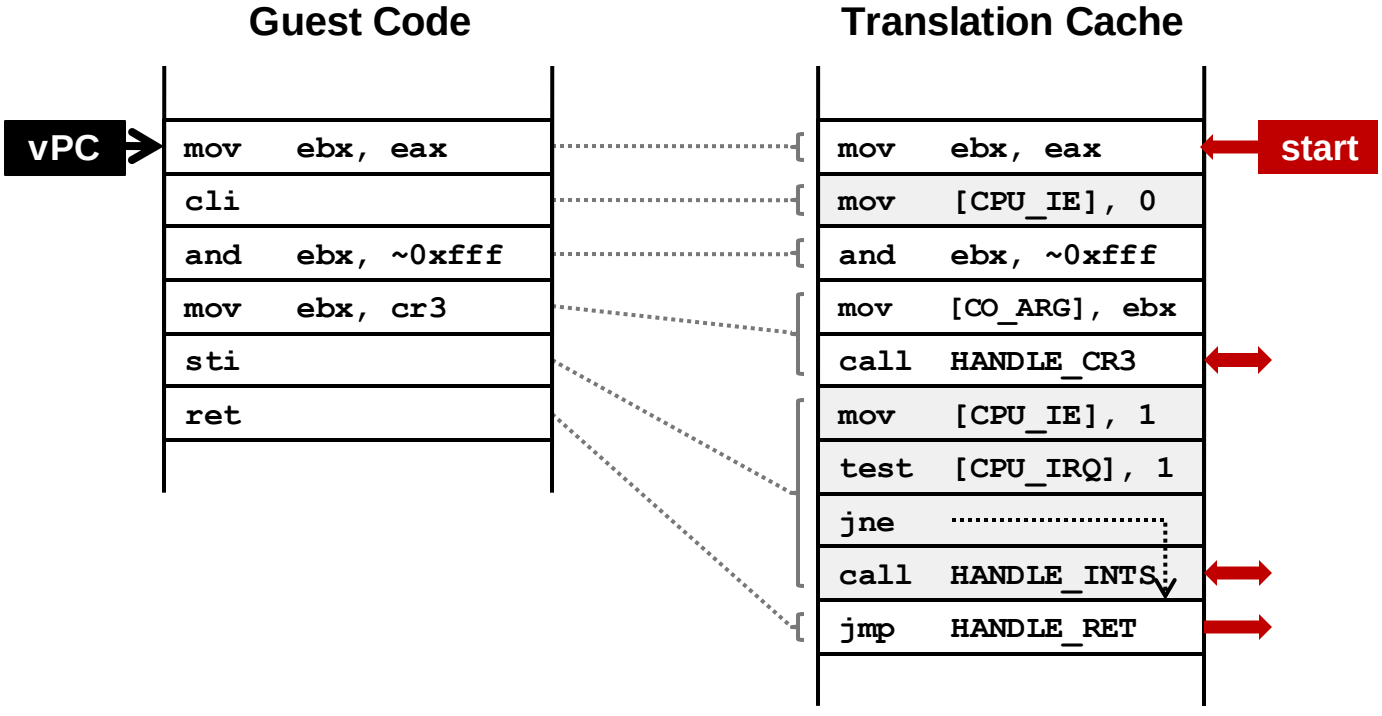
二进制翻译的 Basic Blocks



二进制翻译举例


Guest Code
Translation Cache
vPC
mov ebx, eax
cli
and ebx, ~0xfff
mov ebx, cr3
sti
ret
mov ebx, eax
call HANDLE_CLI
and ebx, ~0xfff
mov [CO_ARG], ebx
call HANDLE_CR3
call HANDLE_STI
jmp HANDLE_RET
start

二进制翻译举例


Guest Code
Translation Cache
vPC
mov ebx, eax
cli
and ebx, ~0xfff
mov ebx, cr3
sti
ret
mov ebx, eax
mov [CPU_IE], 0
and ebx, ~0xfff
mov [CO_ARG], ebx
call HANDLE_CR3
mov [CPU_IE], 1
test [CPU_IRQ], 1
jne
call HANDLE_INTS
jmp HANDLE_RET
start

二进制翻译的翻译器（Translator）-1

```
void BT_Run(void)
{
   CPUState.PC = _start;
   BT_Continue();
}

void BT_Continue(void)
{
   void *tcpc;

   tcpc = BTFindBB(CPUState.PC);

   if (!tcpc) {
      tcpc = BTTranslate(CPUState.PC);
   }

   RestoreRegsAndJump(tcpc);
}
```

```
void *BTTranslate(uint32 pc)
{
   void *start = TCTop;
   uint32 TCPC = pc;

   while (1) {
      inst = Fetch(TCPC);
      TCPC += 4;

      if (IsPrivileged(inst)) {
         EmitCallout();
      } else if (IsControlFlow(inst)) {
         EmitEndBB();
         break;
      } else {
         /* ident translation */
         EmitInst(inst);
      }
   }

   return start;
}
```

二进制翻译的翻译器（Translator）-2

```
void BT_CalloutSTI (BTSavedRegs regs)
{
   CPUState.PC = BTFindPC(regs.tcpc);
   CPUState.GPR[] = regs.GPR[];

   CPU_STI();

   CPUState.PC += 4;

   if (CPUState.IRQ
         && CPUState.IE) {
      CPUVector();
      BT_Continue();
      /* NOT_REACHED */
   }

   return;
}
```

二进制翻译的缺点

- **不能处理自修改的代码(Self-modifying Code)**
- **中断插入粒度变大**
  - 模拟执行可以在任意指令位置插入虚拟中断
  - 二进制翻译时只能在基本块边界插入虚拟中断（为什么？）

方法1&2总结

- **指令执行：软件模拟虚拟化**
- **解释执行**
  - 将虚拟机所需执行的每一条指令都经由 VMM 实时解释执行
  - 虚拟机的每一条指令都被 VMM 解释成能够在宿主机运行的一个函数
  - 开源模拟器 Bochs
- **扫描与修补**
  - 保留普通指令
  - 修补敏感指令
- **二进制翻译**

方法3：半虚拟化 (Para-virtualization)

- **协同设计**
  - 让VMM提供接口给虚拟机，称为**Hypercall**
  - 修改操作系统源码，让其主动调用VMM接口
- **Hypercall可以理解为VMM提供的系统调用**
  - x86中VMCALL指令、在ARM中HVC指令、RISC-V中HLV指令
- **将所有不引起下陷的敏感指令替换成超级调用**
- **思考：这种方式有什么优缺点?**

半虚拟化方法的优缺点

- **优点：**
  - 解决了敏感函数不下陷的问题
  - 协同设计的思想可以提升某些场景下的系统性能
    - I/O等场景
- **缺点：**
  - 需要修改操作系统代码，难以用于闭源系统
  - 即使是开源系统，也难以同时在不同版本中实现

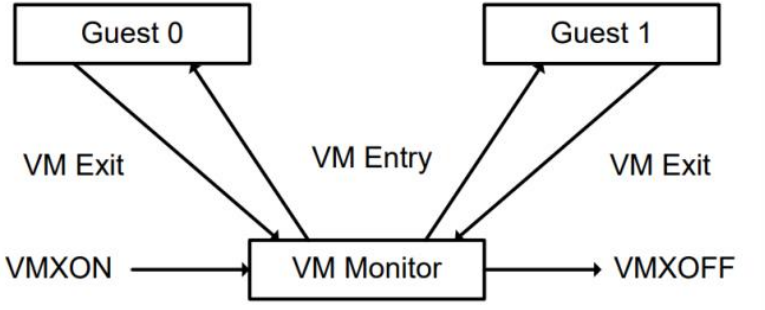
方法4：硬件虚拟化

- **x86和RISC-V都引入了全新的虚拟化特权级**
- **x86引入了root模式和non-root模式**
  - Intel推出了VT-x硬件虚拟化扩展
  - Root模式是最高特权级别，控制物理资源
  - VMM运行在root模式，虚拟机运行在non-root模式
  - 两个模式内都有4个特权级别：Ring0~Ring3
- **RISC-V引入了HS、VS和VU模式**
  - VMM运行在HS模式下
  - M模式是最高特权级别，控制物理资源
  - VM的操作系统和应用程序分别运行在VS和VU模式

INTEL VT-X

VT-x的处理器虚拟化

- **Intel VT**
  - 通过 VMX(Virtual Machine Extension) 操作模式修补虚拟化漏洞
- **两种操作模式**
  - 根操作模式
  - 非根操作模式
  - 每种模式都支持Ring 0 ~ Ring 3



VT-x的处理器虚拟化


虚拟机N
进程
操作系统内核
Non-root 模式
Ring-3
Non-root 模式
Ring-0
Root模式
Ring-3
进程
Root模式
Ring-0
虚拟机监控器

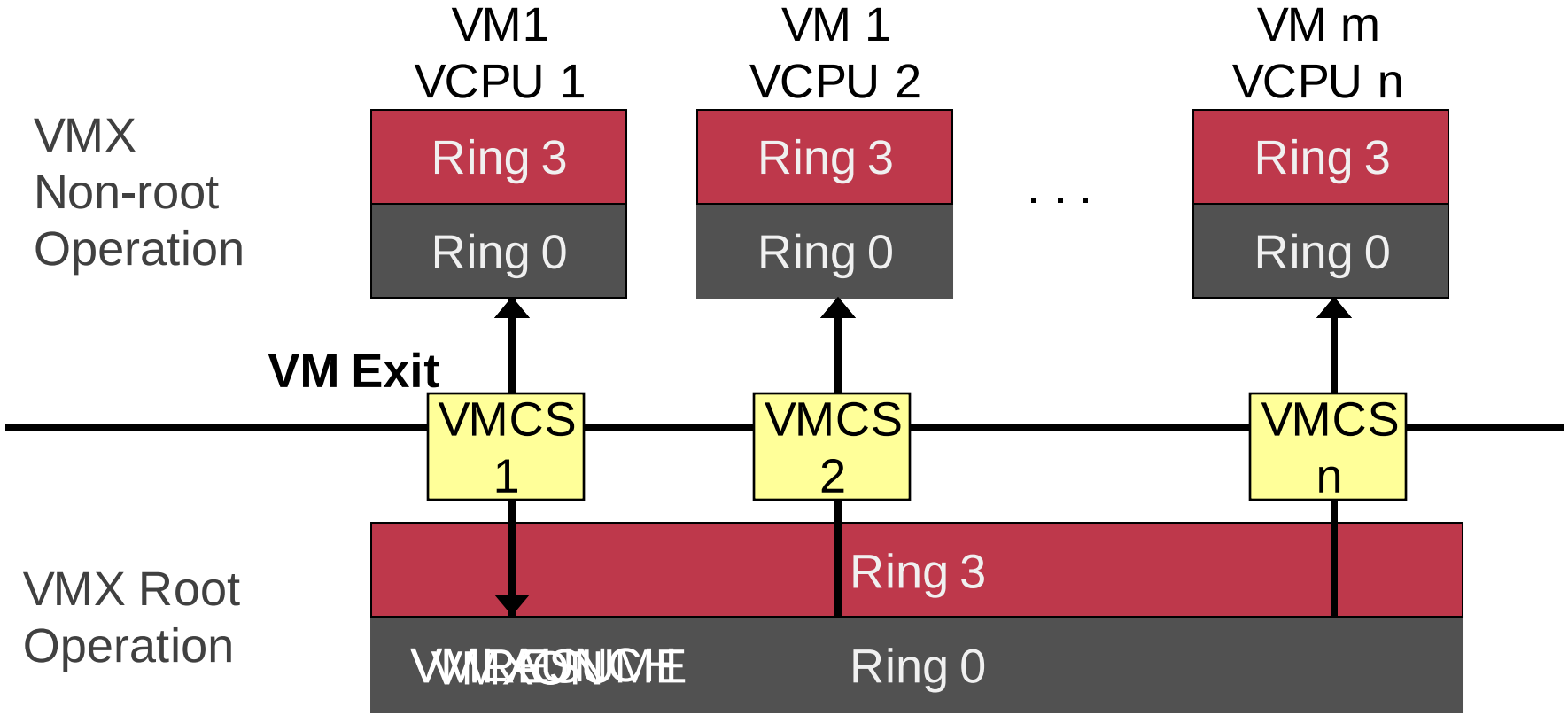
Virtual Machine Control Structure (VMCS)

- **VMM提供给硬件的内存页（4KB）**
  - 记录与当前VM运行相关的所有状态
- **VM Entry**
  - 硬件自动将当前CPU中的VMM状态保存至VMCS
  - 硬件硬件自动从VMCS中加载VM状态至CPU中
- **VM Exit**
  - 硬件自动将当前CPU中的VM状态保存至VMCS
  - 硬件自动从VMCS加载VMM状态至CPU中

VT-x VMCS的内容

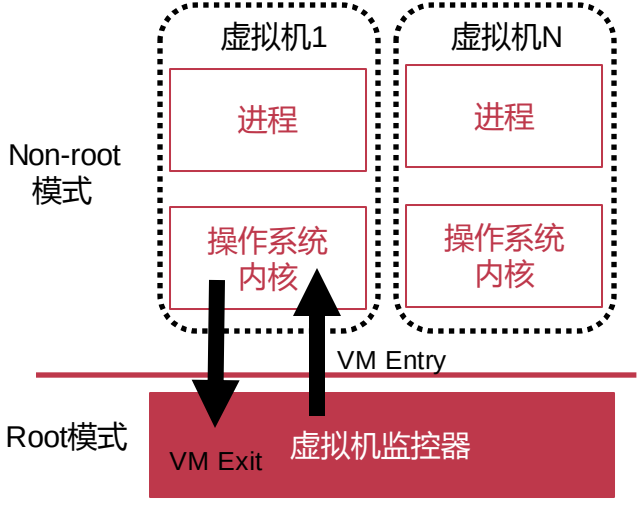
- **包含6个部分**
  - Guest-state area: 发生VM exit时，CPU的状态会被硬件自动保存至该区域；发生VM Entry时，硬件自动从该区域加载状态至CPU中
  - Host-state area：发生VM exit时，硬件自动从该区域加载状态至CPU中；发生VM Entry时，CPU的状态会被自动保存至该区域
  - VM-execution control fields： 控制Non-root模式中虚拟机的行为
  - VM-exit control fields：控制VM exit的行为
  - VM-entry control fields：控制VM entry的行为
  - VM-exit information fields：VM Exit的原因和相关信息（只读区域）

VT-x的执行过程


VM1
VCPU 1
VM 1
VCPU 2
VM m
VCPU n
VMX
Non-root
Operation
Ring 3
Ring 0
Ring 3
Ring 0
. . .
Ring 3
Ring 0
VM Exit
VMCS
1
VMCS
2
VMCS
n
VMX Root
Operation
Ring 3
Ring 0

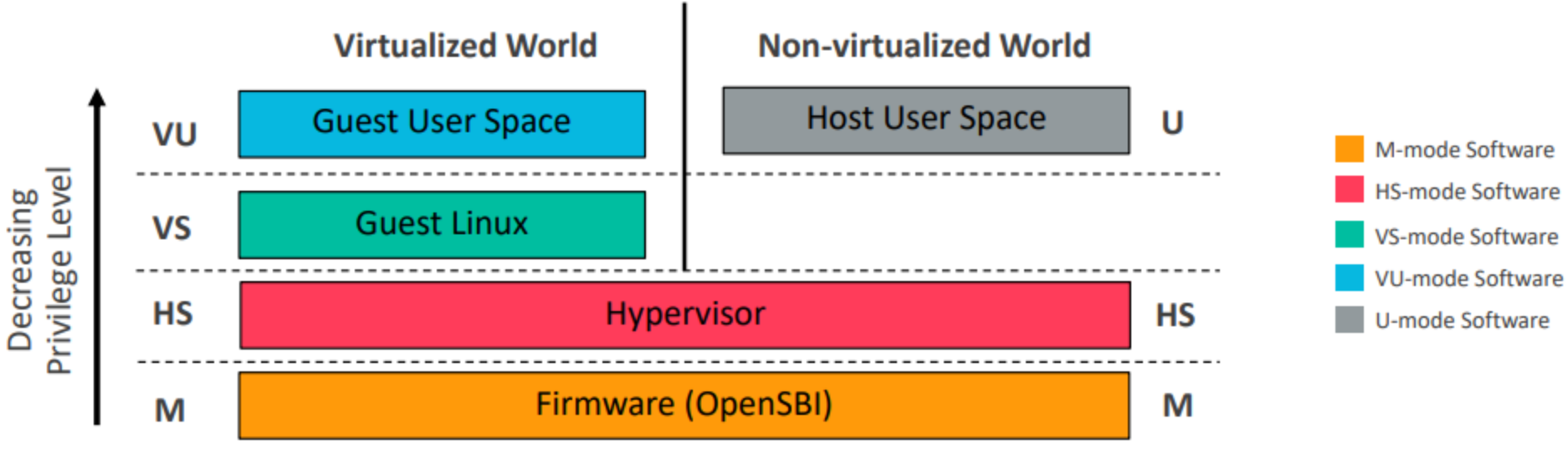
x86中的VM Entry和VM Exit

- **VM Entry**
  - 从VMM进入VM
  - 从Root模式切换到Non-root模式
  - 第一次启动虚拟机时使用VMLAUNCH指令
  - 后续的VM Entry使用VMRESUME指令
- **VM Exit**
  - 从VM回到VMM
  - 从Non-root模式切换到Root模式
  - 虚拟机执行敏感指令或发生事件(如外部中断)



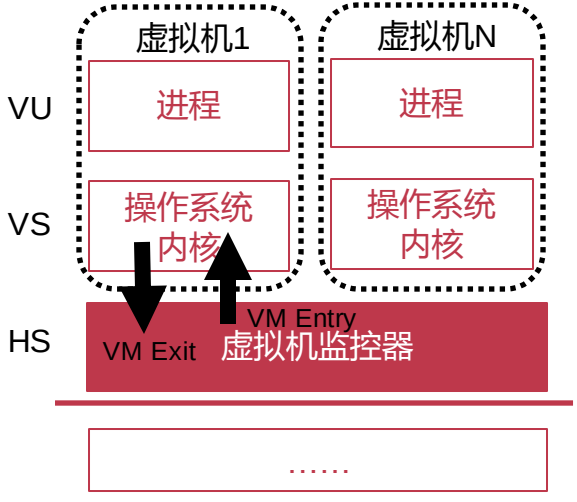
RISC-V的虚拟化技术

RISC-V的处理器虚拟化



RISC-V的VM Entry和VM Exit

- **VM Entry**
  - 使用SRET指令从VMM进入VM
  - 在进入VM之前，VMM需要主动加载VM状态
    - VM内状态：通用寄存器、CSR
    - VM的控制状态：hstatus、hgatp等
- **VM Exit**
  - 虚拟机发生trap或收到中断等
  - 以Exception、Interrupt的形式回到VMM
    - 调用VMM记录在stvec中的处理函数
  - 下陷第一步：VMM主动保存所有VM的状态

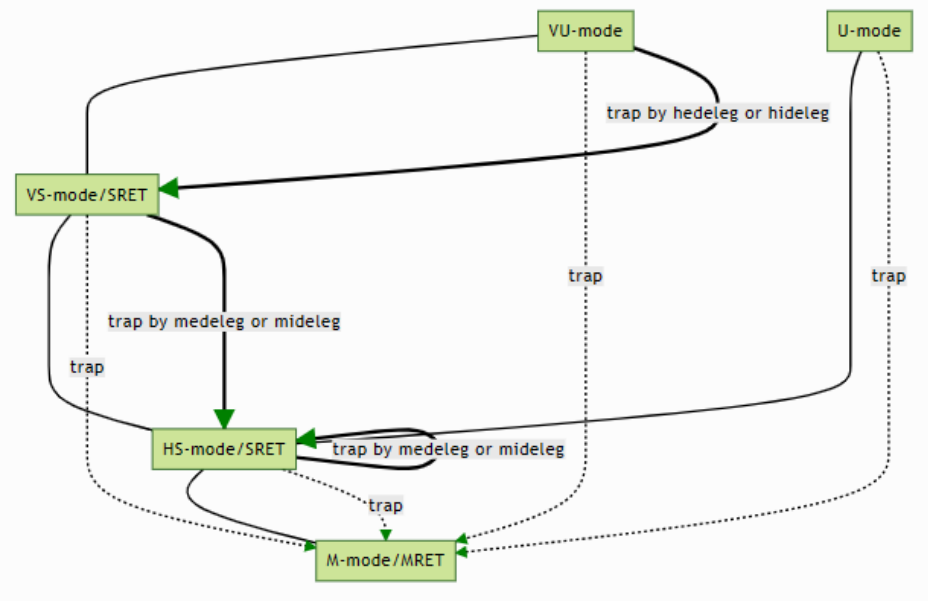


RISC-V：trap 可能导致控制转移及模式切换

正常情况下 trap 都会导致 hart 的控制转移至 M-mode，处理之后通过 mret 指令返回到原来的模式。

特殊情况下 trap 会经由 mdeleg 或 mideleg 委派从 HS-mode 或 VS-mode 转移至 HS-mode，或再经由 hedeleg 或 hideleg 委派从 VU-mode 转移至 VS-mode。

被委派至 HS-mode 和 VS-mode 的 trap 在处理完毕后，将通过 sret 指令返回至 trap 之前的模式。



RISC-V硬件虚拟化的新功能

- **RISC-V中没有VMCS**
- **VM能直接控制VS和VU模式的状态**
  - 可以直接读写satp/sstatus/...
  - 访问satp/sstatus/...实际访问的是vsatp/vsstatus/...，并受这些 vs 寄存器控制
- **VM Exit时VMM可以访问vsatp/vsstatus/...**

- **思考题1：为什么RISC-V中可以不需要VMCS?**
- **思考题2：RISC-V中没有VMCS，对于VMM的设计和实现来说有什么优缺点?**

Hypervisor CSR 简介

- **VMM控制VM行为的系统寄存器**
  - VMM有选择地决定VM在某些情况时下陷
  - 和VT-x VMCS中VM-execution control area类似
- **在VM Entry之前设置相关位，控制虚拟机行为**
  - hstatus.TVM：VM执行虚拟内存操作是否下陷，如读写satp，sfence.vma
  - hstatus.TW：执行WFI指令是否下陷
  - hedeleg/hideleg：Exception/Interrupt委托还是下陷
  - hgatp: 控制第二阶段地址翻译

Hypervisor CSR简介

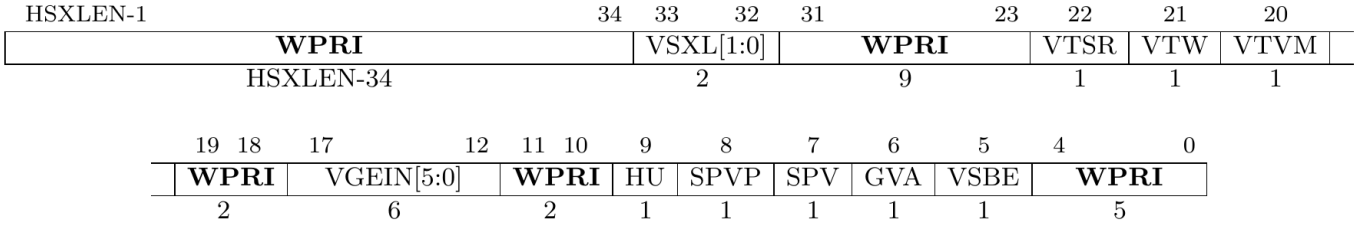


Figure 8.2: Hypervisor status register (`hstatus`) when HSXLEN=64.

Figure 8.3: Hypervisor exception delegation register (`hedeleg`).

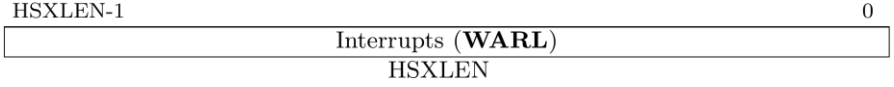
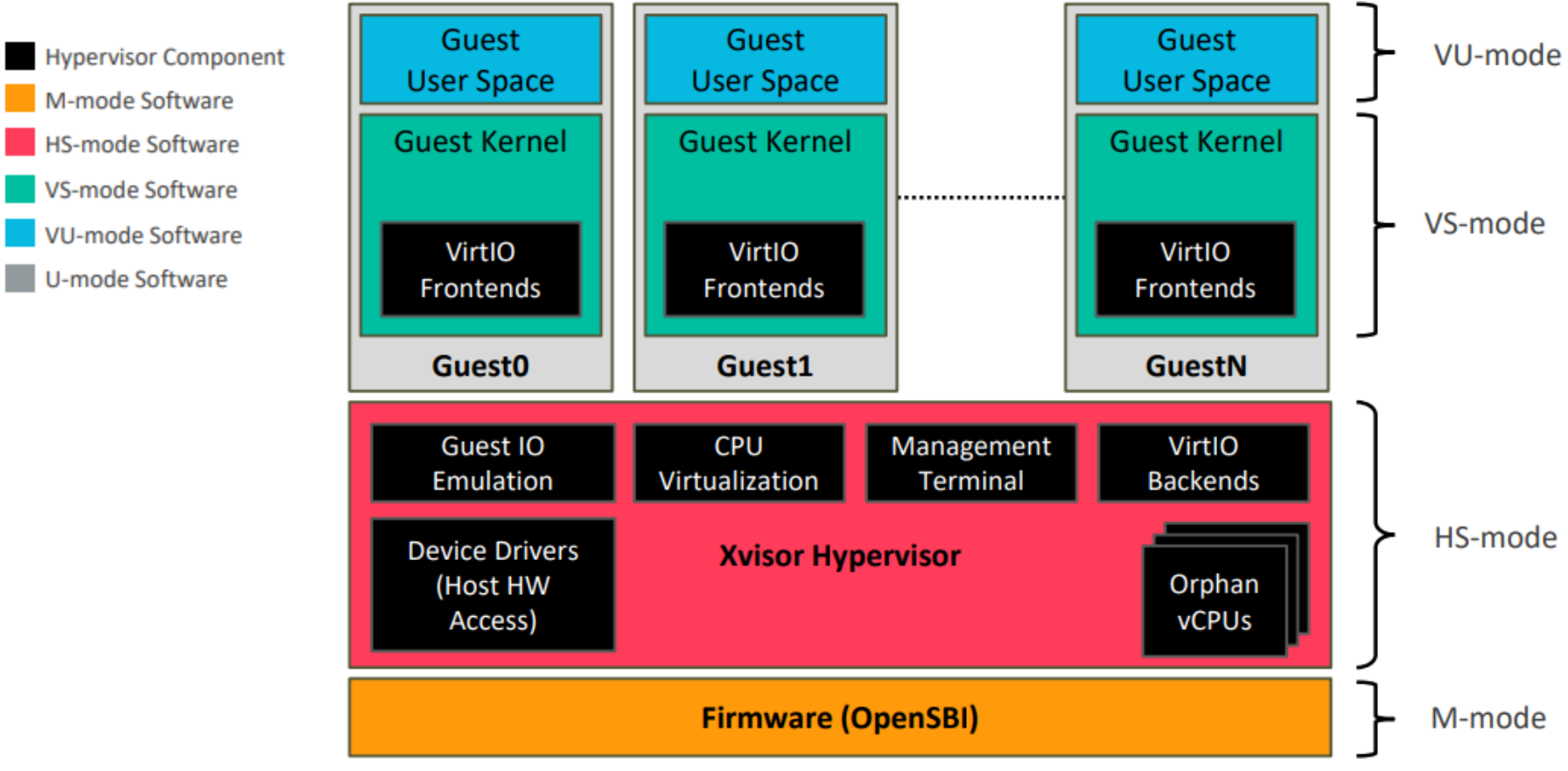


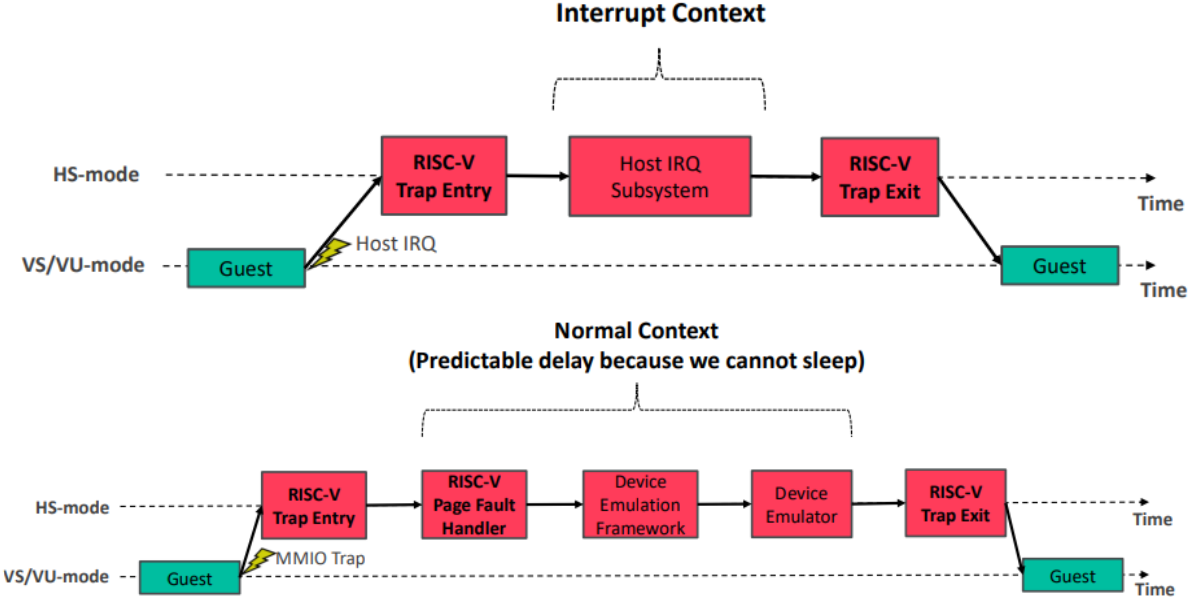
Figure 8.4: Hypervisor interrupt delegation register (`hideleg`).

RISC-V中的Type-1 VMM架构



RISC-V：trap处理

- 处理主机中断
- 处理虚拟机中的虚拟 CPU 的 MMIO trap



VT-x和RV64H对比

	VT-x	RV64H
新特权级	Root和Non-root	HS、VS和VU
是否有VMCS?	是	否
VM Entry/Exit时硬件自动保存状态?	是	否
是否引入新的指令?	是(多)	是(少)
是否引入新的系统寄存器?	否	是(多)
是否有扩展页表(第二阶段页表)?	是	是

Type-1和Type-2在VT-x和RV64H下架构