

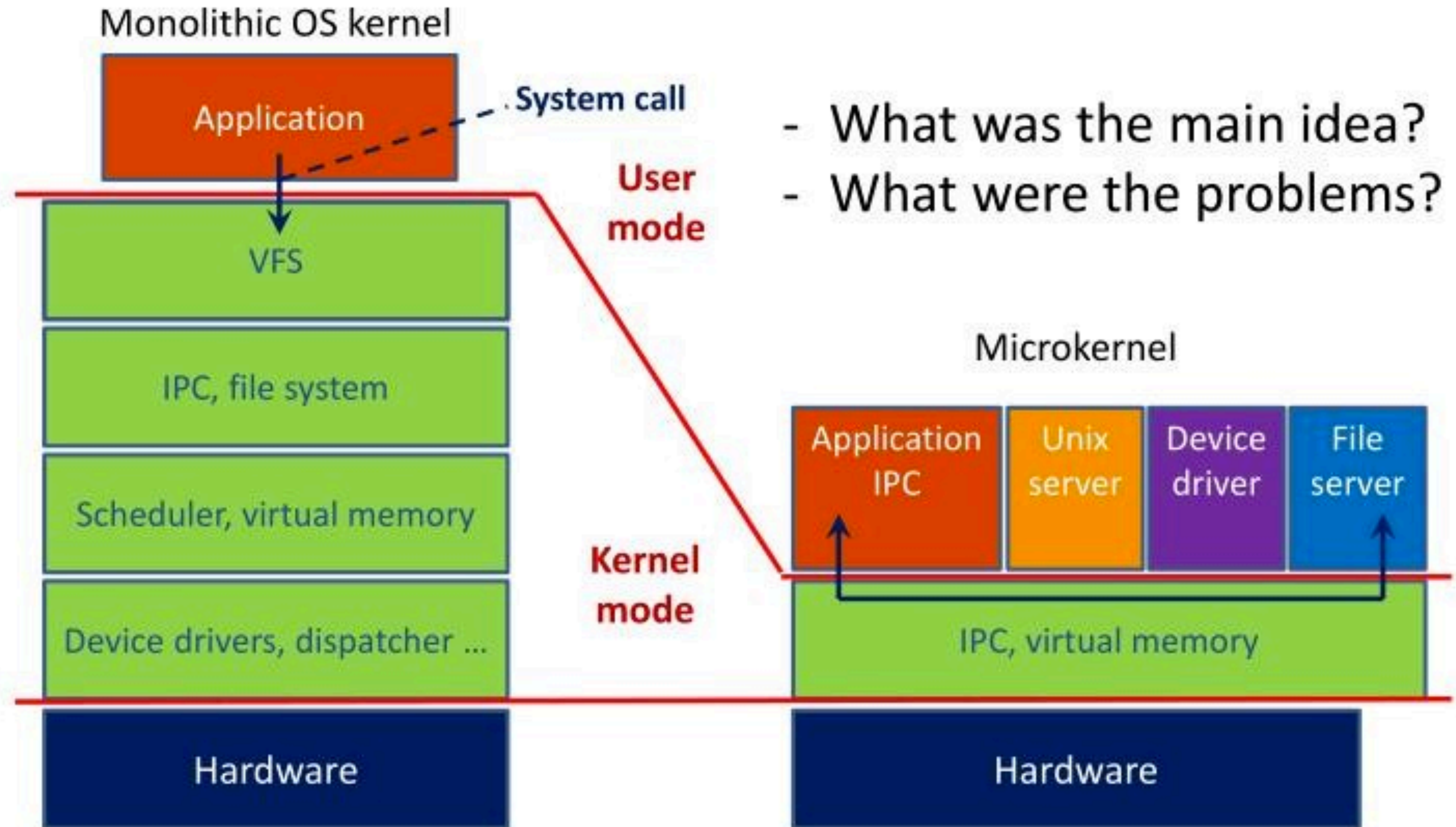
Virtualization Basics and Virtualizing CPU

Yiying Zhang

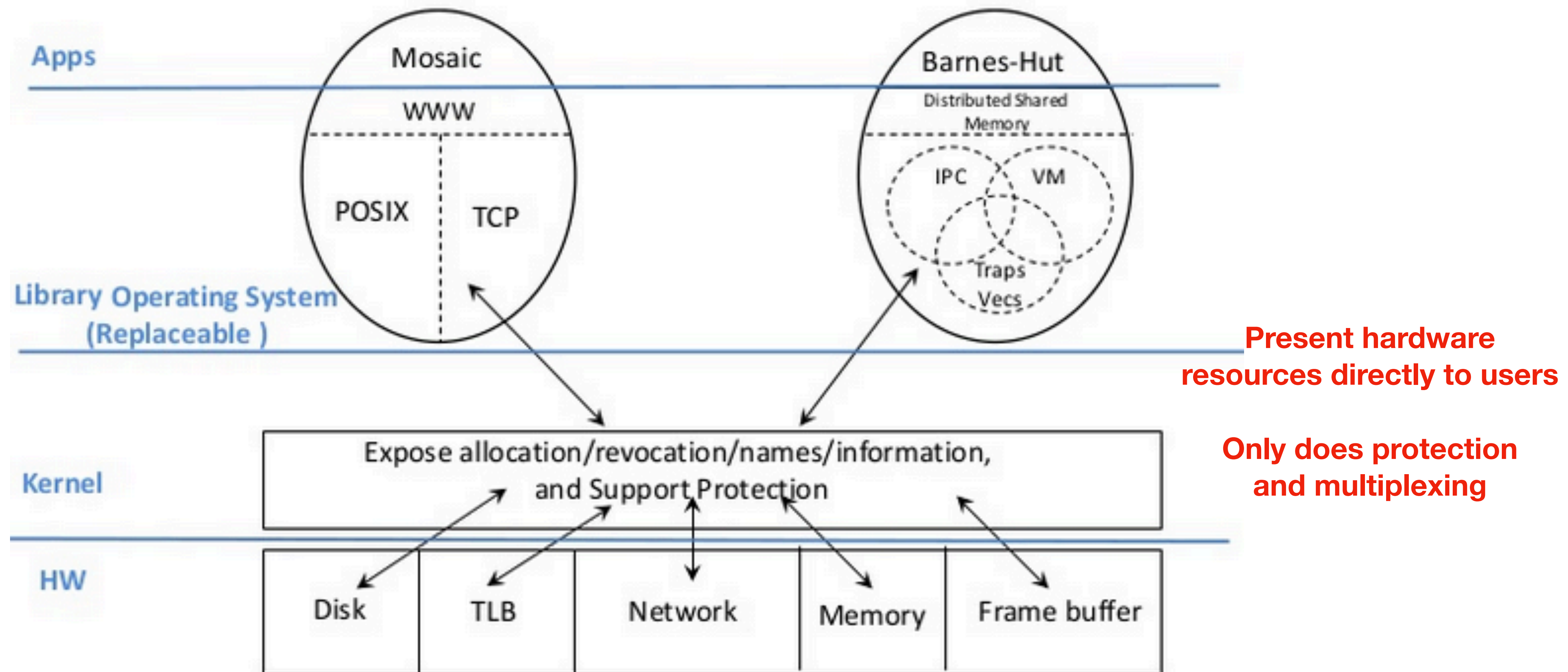
Administravia

- Form project group by next Wed (4/10)
 - email TA (or me, if we don't get a TA by the end of this week) your group info (name, student ID, email address, chosen project topic)
 - If you can't decide on a project topic by 4/10, just send the other info
- Class attendance tracked from next Wed

Monolithic kernel vs Microkernel



ExoKernel



OS Functionalities

- Process management
 - Virtual memory system
 - File and storage system
-
- Networking
 - Other I/O systems
 - Command-interpreter system

Virtualized Functionalities

- Virtualize sensitive instructions
 - Virtualized physical memory
 - Virtual disk
-
- Virtual network interface
 - Other virtualized I/O systems
 - Command-interpreter system

Virtualization Principles

Popek and Goldberg's virtualization principles in 1974:

- ***Fidelity***. Software on the VMM executes identically to its execution on hardware, barring timing effects.
- ***Performance***. An overwhelming majority of guest instructions are executed by the hardware without the intervention of the VMM.
- ***Safety***. The VMM manages all hardware resources.

Virtualization Approach 1: Complete Machine Emulation (Hosted Interpretation)

- VMM implements the complete hardware architecture in software
- VMM steps through VM's instructions and update emulated hardware as needed

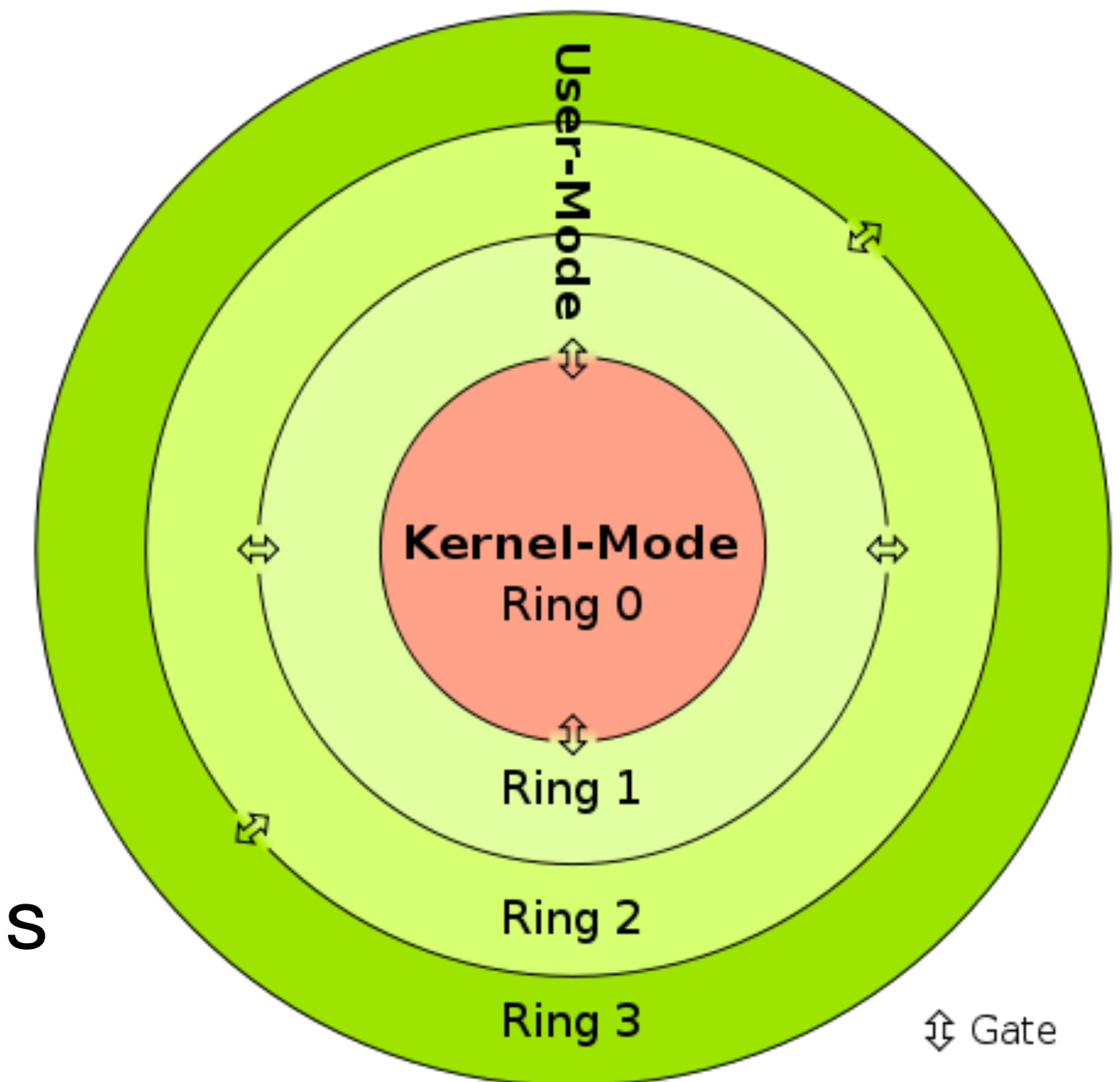
```
while(1){  
    curr_instr = fetch(virtHw.PC);  
    virtHw.PC += 4;  
    switch(curr_instr){  
        case ADD:  
            int sum = virtHw.regs[curr_instr.reg0] +  
                      virtHw.regs[curr_instr.reg1];  
            virtHw.regs[curr_instr.reg0] = sum;  
            break;  
        case SUB:  
            //...etc...
```

Complete Machine Emulation (Hosted Interpretation)

- Pros
 - Easy to handle all types of instructions (can enforce policy when doing so)
 - Provides complete isolation (no guest instructions runs directly on hardware)
 - Can debug low-level code (e.g., boot code) in the guest
- Cons
 - Emulate a modern processor is difficult
 - Violates performance requirement (*it is really slow!*)

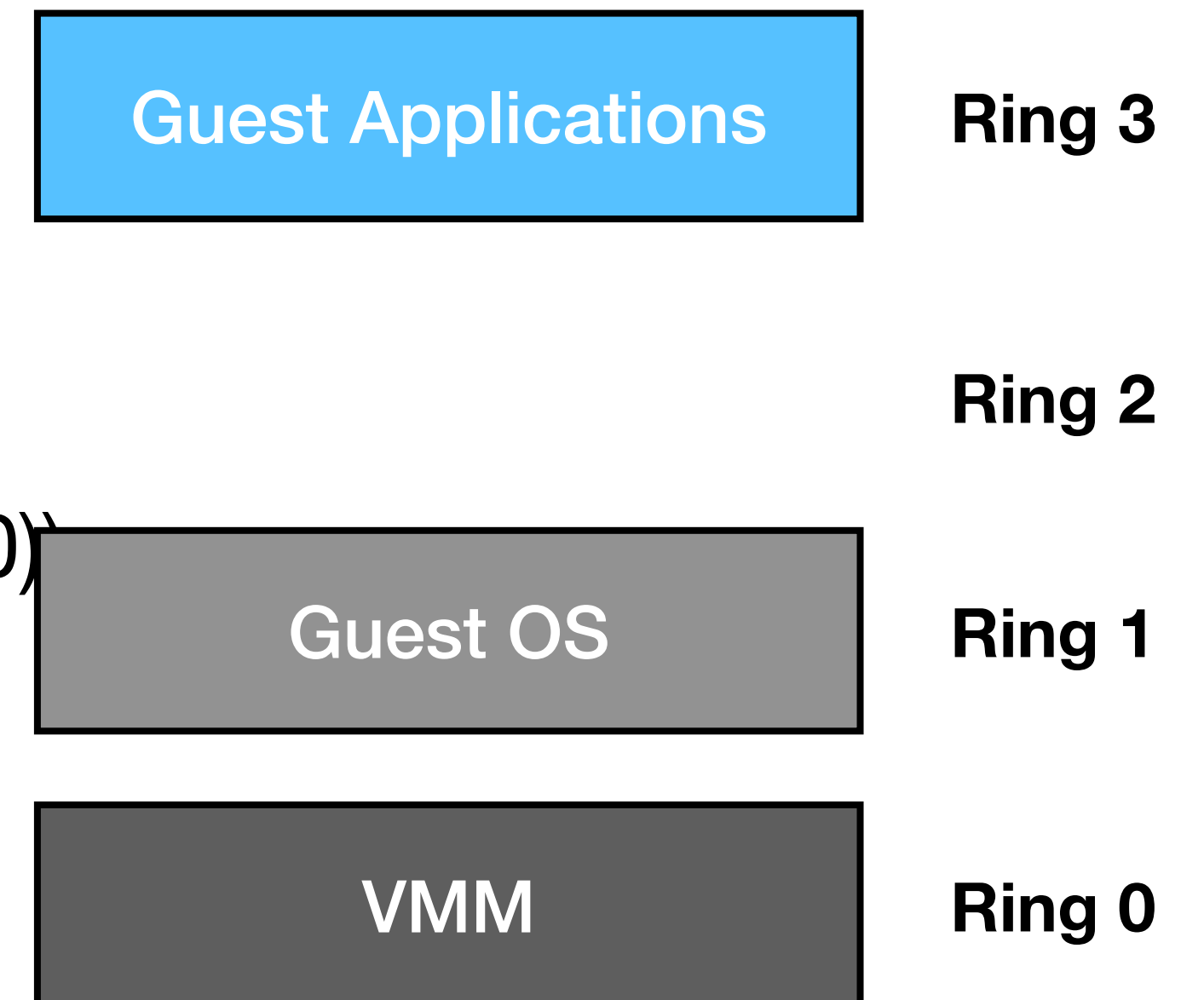
Protection Rings

- More privileged rings can access memory of less privileged ones
- Calling across rings can only happen with hardware enforcement
- Only Ring 0 can execute privileged instructions
- Rings 1, 2, and 3 trap when executing privileged instructions
- Usually, the OS executes in Ring 0 and applications execute in Ring 3



Virtualization Approach 2: Direct Execution with Trap-and-Emulate

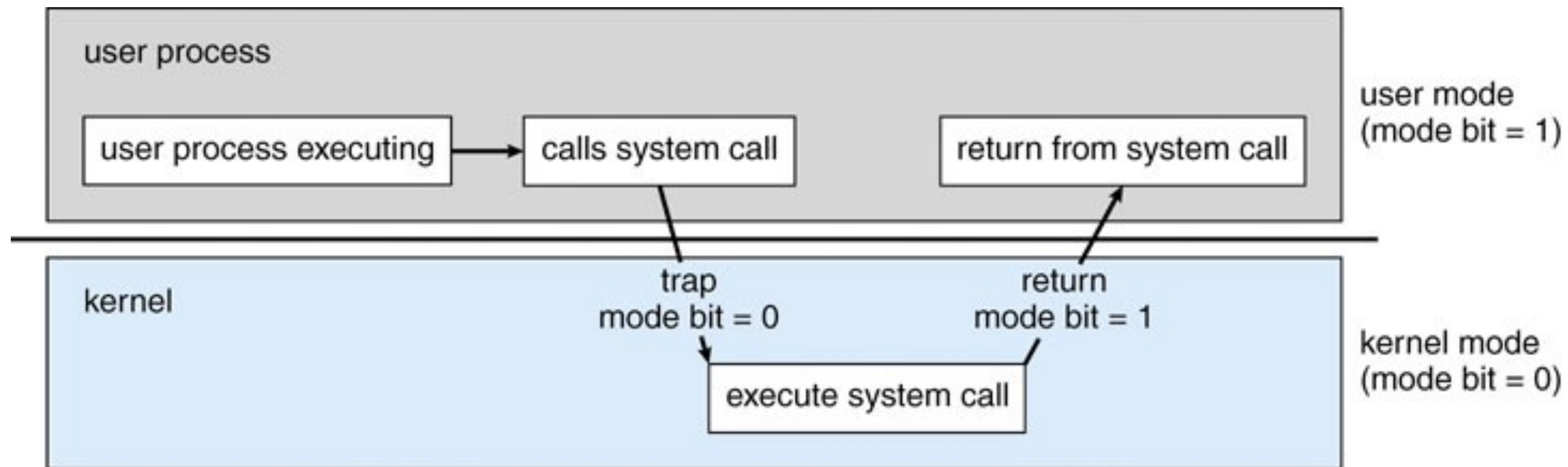
- Idea: execute most guest instructions natively on hardware (assuming guest OS runs on the same architecture as real hardware)
- Applications run in ring 3 (can't access memory owned by guest OS (ring1))
- Guest OS runs in ring 1 (can't access memory owned by VMM (ring 0))
- Cannot allow guest OS to run *sensitive instructions* directly!
 - Those that touches hardware configurations



Trap-and-Emulate

- Goal: hand off sensitive operations to the VMM
- VMM emulates the effect of sensitive operations on virtual hardware provided to the guest OS
 - VMM controls how the VM interacts with physical hardware
 - VMM fools the guest OS into thinking that it runs at the highest privilege level

(Background) Transition between User/Kernel Modes



fg1_10

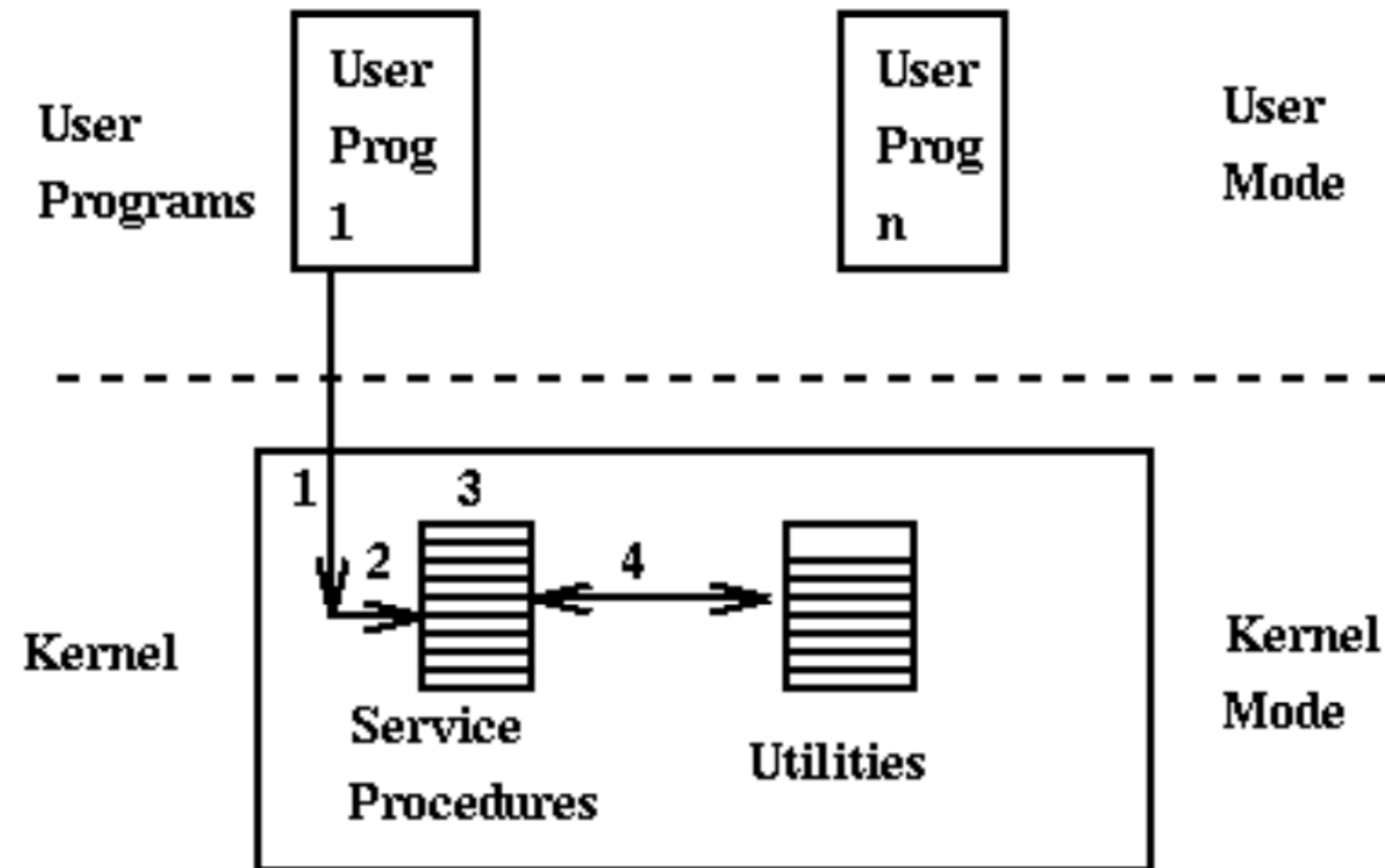
(Background) Interrupt

- A mechanism for coordination between concurrently operating units of a computer system (e.g. CPU and I/O devices) to respond to specific conditions within a computer
- Results in transfer of flow of control (to interrupt handler in the OS), forced by hardware
- Hardware interrupts
 - I/O devices: NIC, keyboard, etc.
 - Timer
- Software interrupts
 - Exception: a software error (e.g., divided by zero)
 - System call

(Background) Handling Interrupts

- Incoming interrupts are disabled (at this and lower priority levels) while the interrupt is being processed to prevent a lost interrupt
- Interrupt architecture must save the address of the interrupted instruction
- Interrupt transfers control to the interrupt service routine
 - generally, through the interrupt vector, which contains the addresses of all the service routines
- If interrupt routine modifies process state (register values)
 - save the current state of the CPU (registers and the program counter) on the system stack
 - restore the state before returning
- Interrupts are re-enabled after servicing current interrupt
- Resume the interrupted instruction

(Background) Flow of a System Call



1. System call (User→Kernel Mode)
 2. Check parameters
 3. Call service routine
 4. Service Routine call utilities
- Reschedule/Return to user

(Background) Regular System Call

Process	Hardware	Operating System
1. Execute instructions (add, load, etc.)		
2. System call: Trap to OS		
	3. Switch to kernel mode; Jump to trap handler	
		4. In kernel mode; Handle system call; Return from trap
	5. Switch to user mode; Return to user code	
6. Resume execution (@PC after trap)		

Figure B.1: Executing a System Call

Example of Trap-and-Emulate: System Calls with Virtualization

Process	Operating System
1. System call: Trap to OS	
	2. OS trap handler: Decode trap and execute appropriate syscall routine; When done: return from trap
3. Resume execution (@PC after trap)	

Figure B.2: System Call Flow Without Virtualization

Process	Operating System	VMM
1. System call: Trap to OS		
		2. Process trapped: Call OS trap handler (at reduced privilege)
	3. OS trap handler: Decode trap and execute syscall; When done: issue return-from-trap	
		4. OS tried return from trap: Do real return from trap
5. Resume execution (@PC after trap)		

Figure B.3: System Call Flow with Virtualization

Trap-and-Emulate

- Pros and Cons?
- Performance implications
 - Almost no overhead for non-privileged instructions
 - Large overhead for privileged instructions

x86 Difficulties

Popek and Goldberg (1974) two classes of instructions

- ***privileged instructions***: those that trap when in user mode
 - ***sensitive instructions***: those that modify or depends on hardware configs
 - A machine can be virtualized (using trap-and-emulate) if every sensitive instruction is privileged.
- Reality: only privileged operations trap to VMM
 - Not all sensitive instructions are privileged with x86 for many years, i.e., non-virtualizable processor
 - These instructions do not trap and behave differently in kernel and user mode
 - Example: ***popf***
 - Pops 16 bits from top of the stack to the ***%eflags*** register
 - Bit 9 of ***%eflags*** masks interrupts (i.e., enables/disables interrupts)
 - ***popf*** is not privileged. What happens if guest OS (ring 1) runs ***popf*** to ***%eflags***?
 - In Ring 0, ***popf*** can set bit 9, but CPU silently ignores ***popf*** when running in Ring 1
 - What should happen is a trap so that VMM can emulate interrupts (change which interrupts to forward to guest OS)

Virtualization Approach 3: Direct Execution with Binary Translation

- VMM dynamically rewrites instructions
- So that non-virtualizable instructions can trap to VMM
- VMware's main selling point (at least in early years)

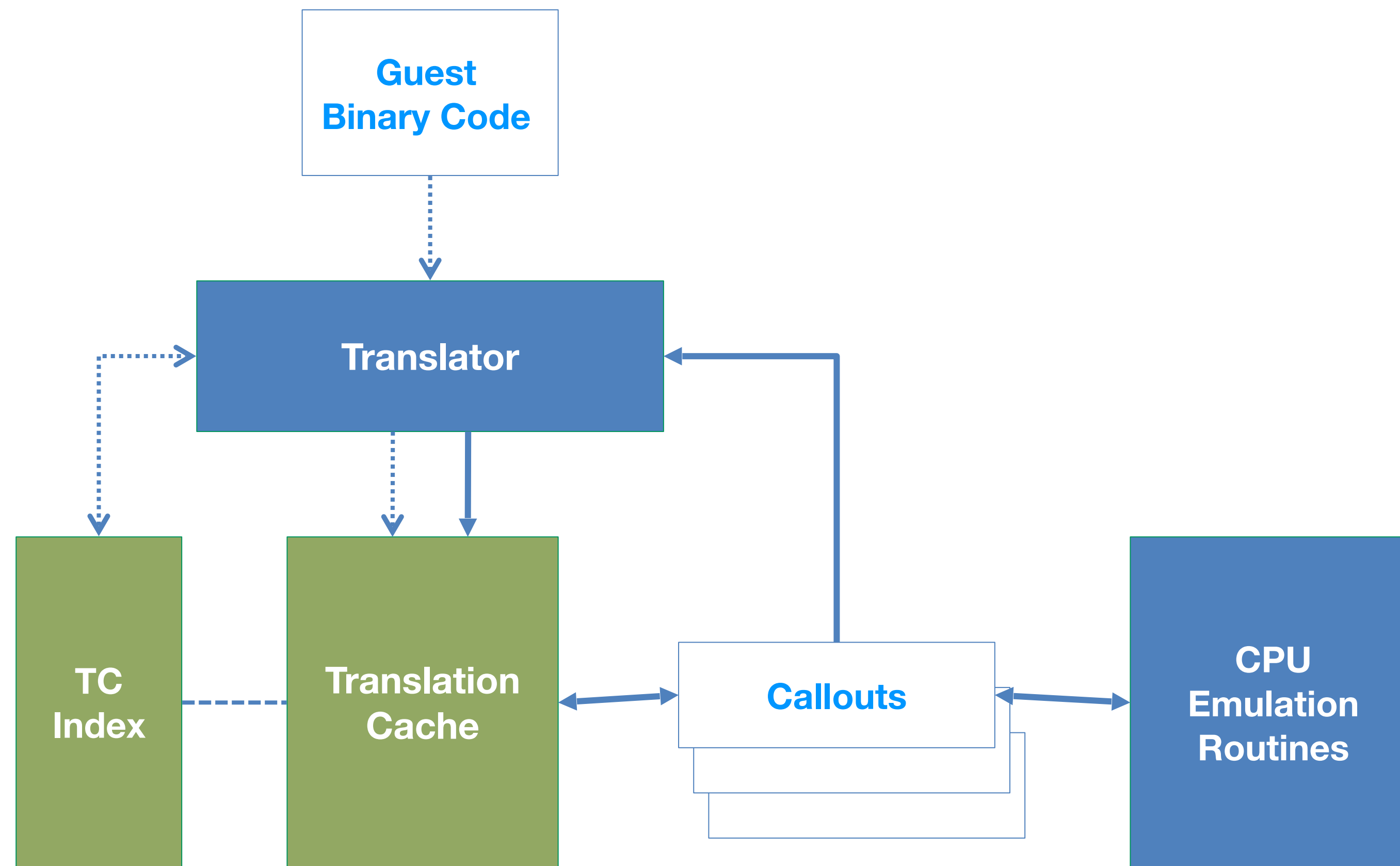
Basic Idea of Binary Translation

- Based on input guest binary, compile (translate) instructions in a cache
- and run them directly
- Challenges:
 - Correctness of direct memory addressing
 - Handling relative memory addressing (e.g., jumps)
 - Handling sensitive instructions
 - Achieving good performance

VMware's Dynamic Binary Translation

- *Binary*: Input is binary x86 code
- *Dynamic*: Translation happens at runtime
- *On demand*: Code is translated only when it is about to execute
- *System level*: Rules set by x86 ISA, not higher-level ABIs
- *Subsetting*: Output a safe subset of input full x86 instruction set
- *Adaptive*: Translated code is adjusted according to guest behavior changes

Architecture of VMware's Binary Translation

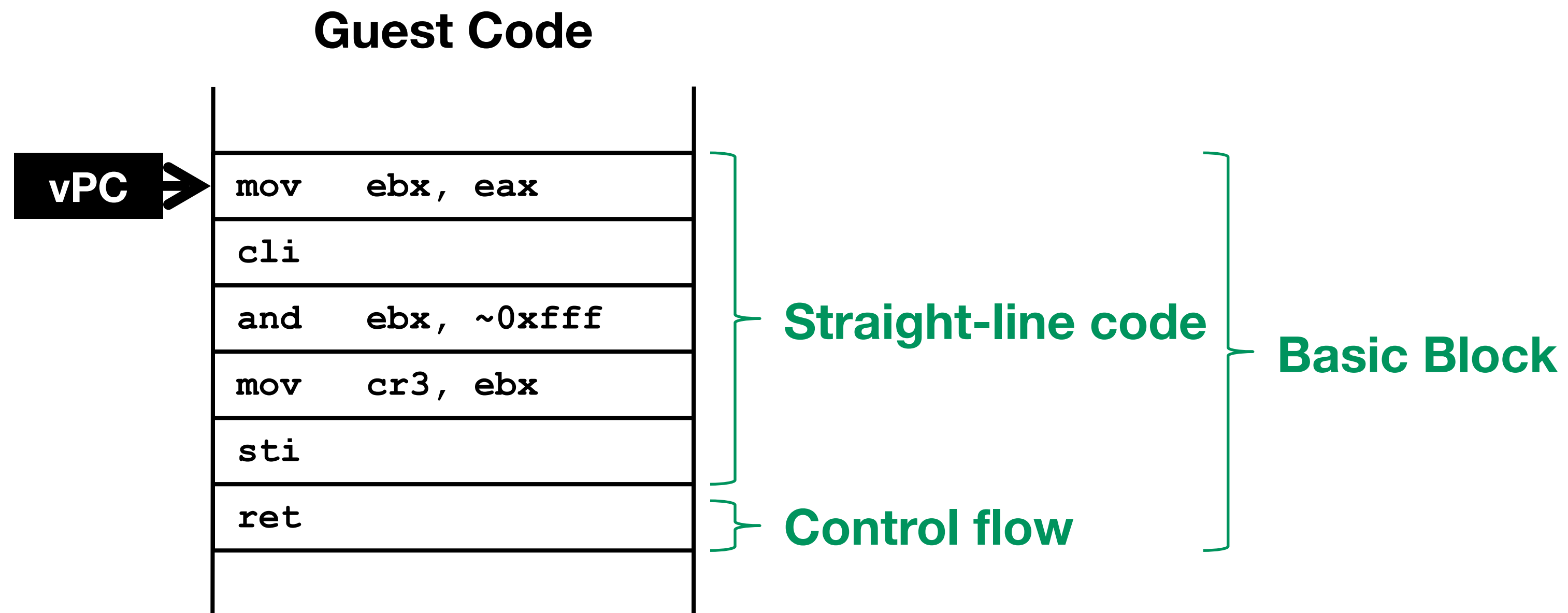


Translation Unit

- TU: 12 instructions or a “terminating” instruction (a basic code block)
- Why TU as the unit not individual instruction or larger code range?
- TU -> Compiled Code Fragment (CCF)
- CCF stored in translation cache (TC)
- At the end of each CCF, call into translator to decide and translate the next TU
 - If the destination code is already in TC, then directly jumps to it
 - Otherwise, compiles the next CCF into TC

implemented by the VMM

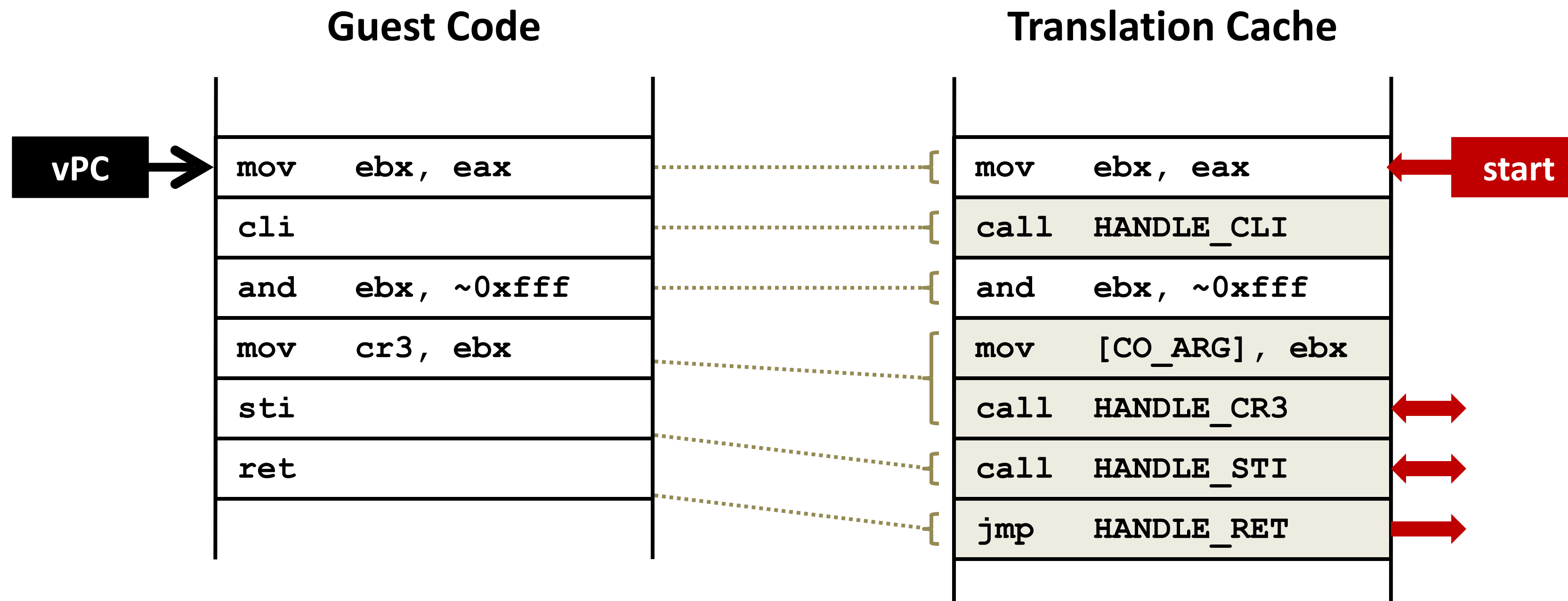
Basic Blocks



IDENT/Non-IDENT Translation

- Most instructions can be translated *IDENT*, except for
 - PC-relative address
 - Direct control flow
 - Indirect control flow
 - Sensitive instructions
 - If already traps, then can be handled when it traps (more optimization soon to be discussed)
 - Otherwise, replace it with a call to the emulation function

Binary Translation



Basic Binary Translator

```
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;
}
```

Example of a CPU emulation function (store interrupt flag):

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

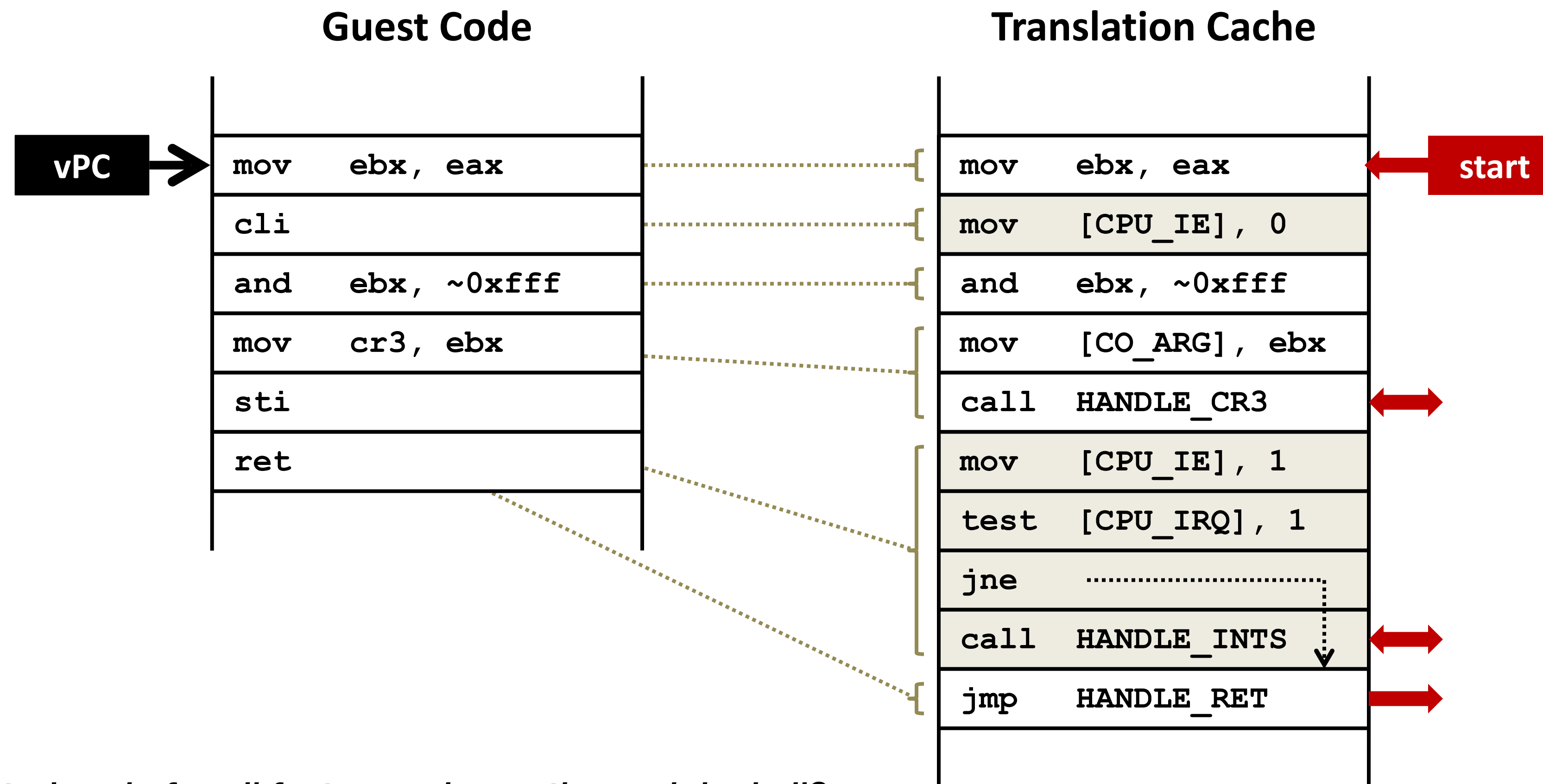
    CPU_STI();

    CPUState.PC += 4;

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

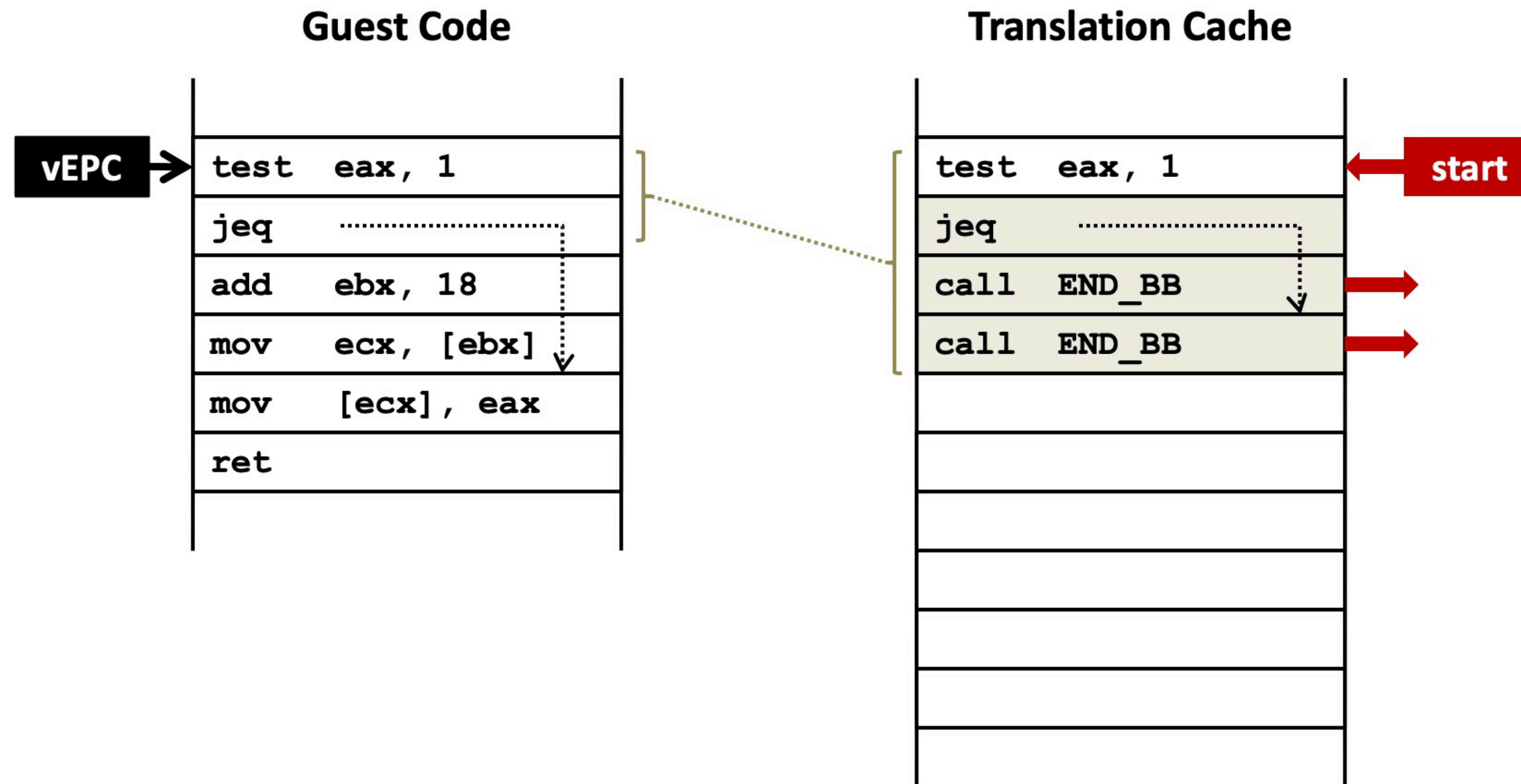
    return;
}
```

Binary Translation

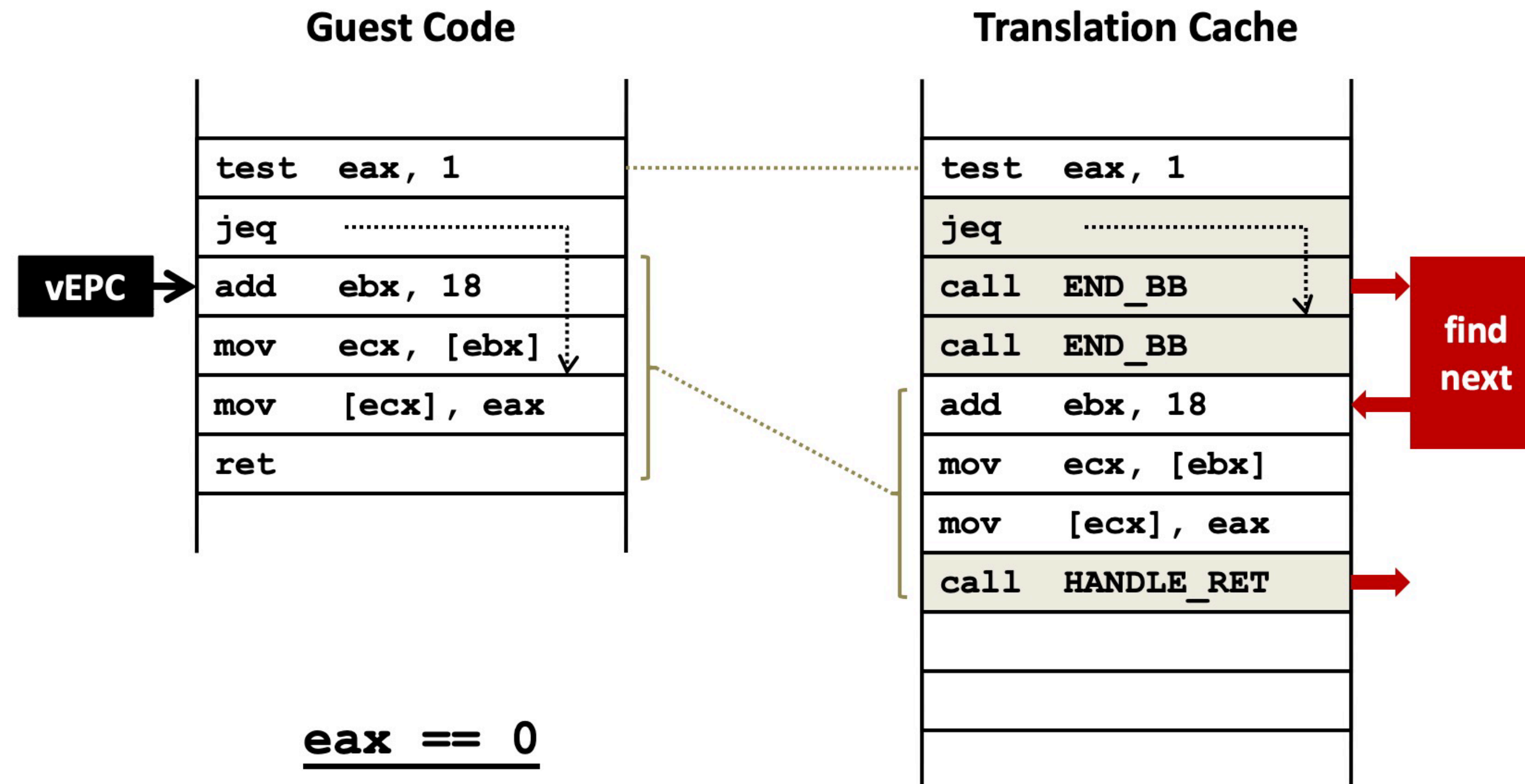


Q: Is the translated code for cli faster or slower than original cli?

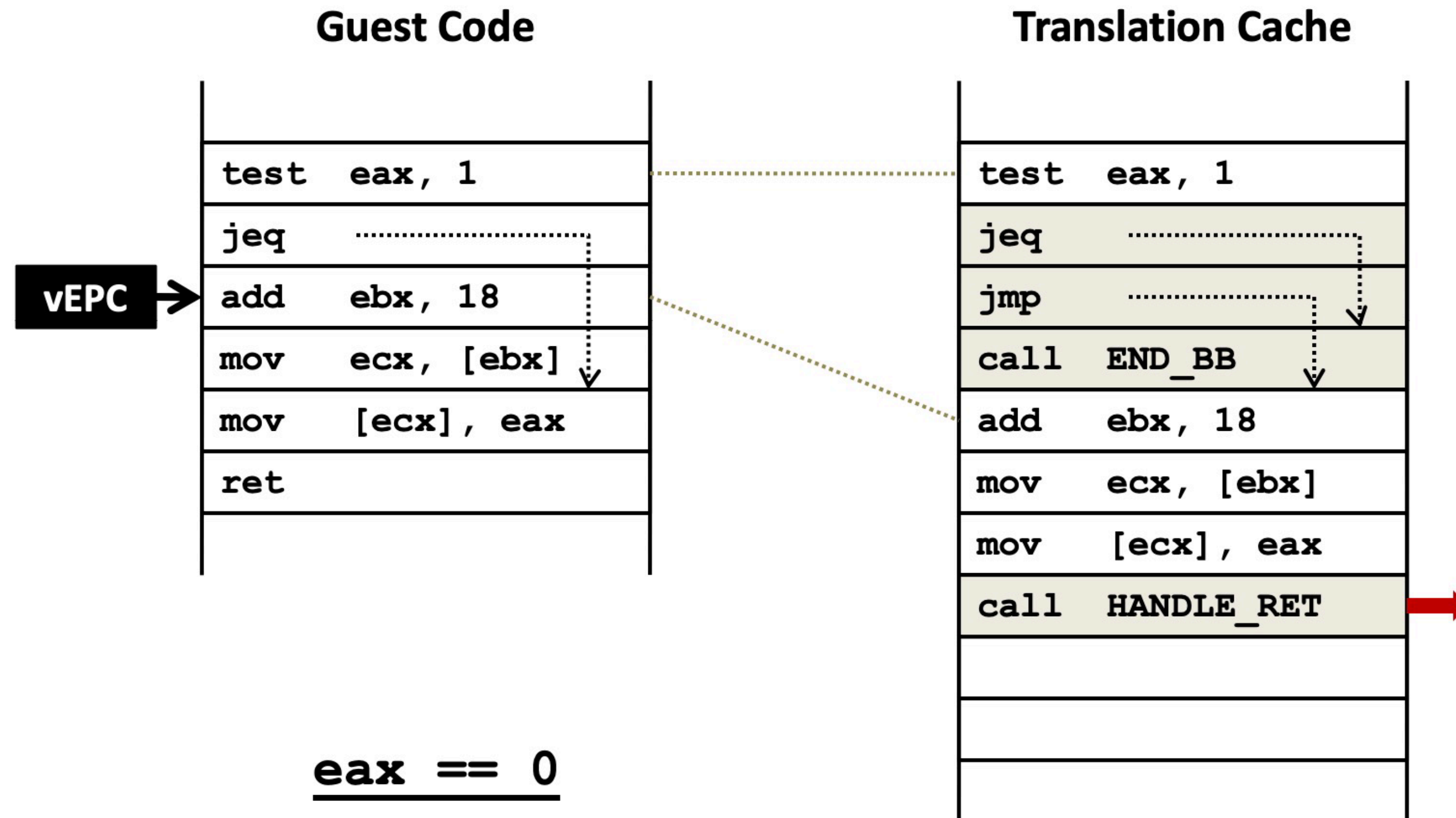
Controlling Control Flow



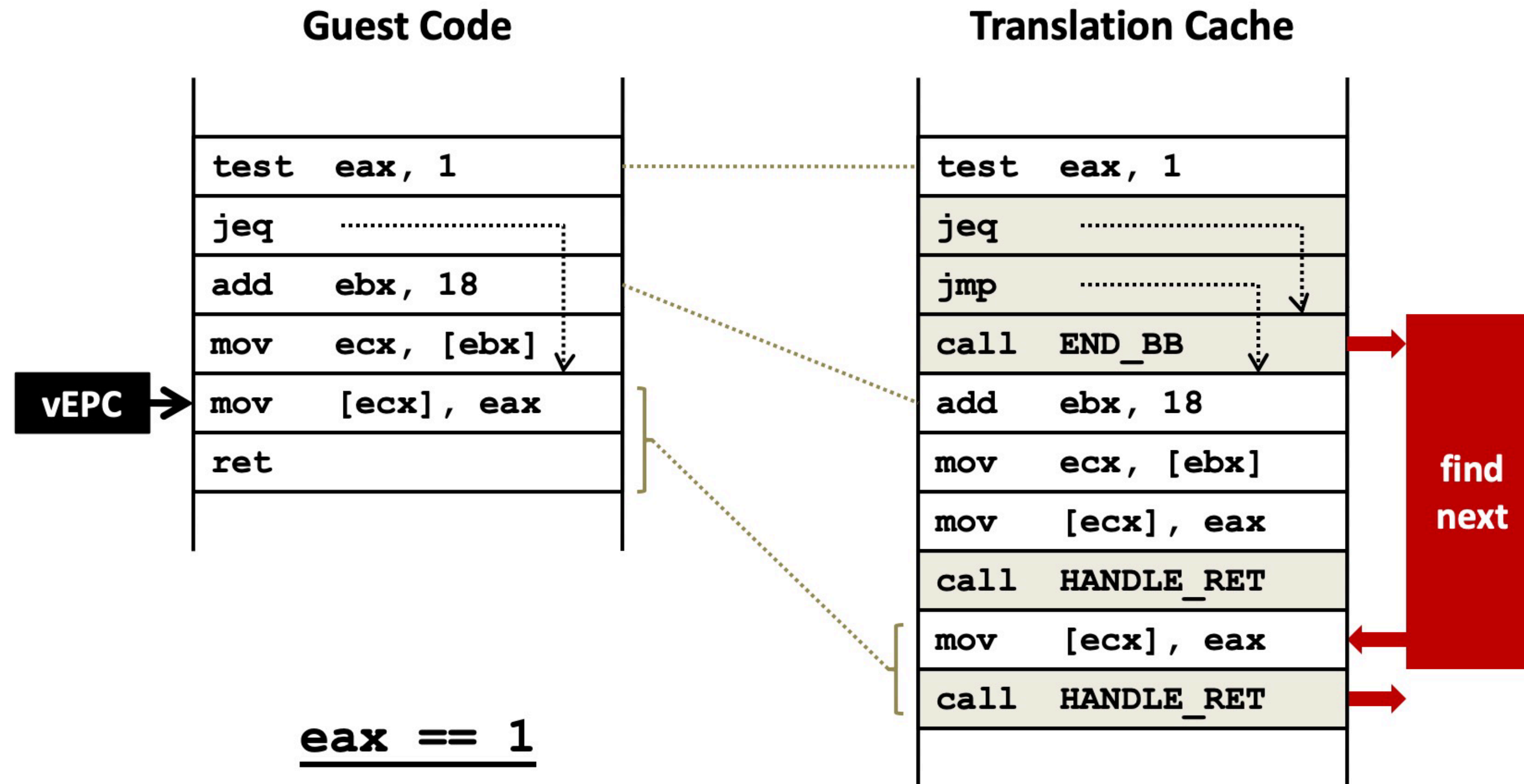
Controlling Control Flow



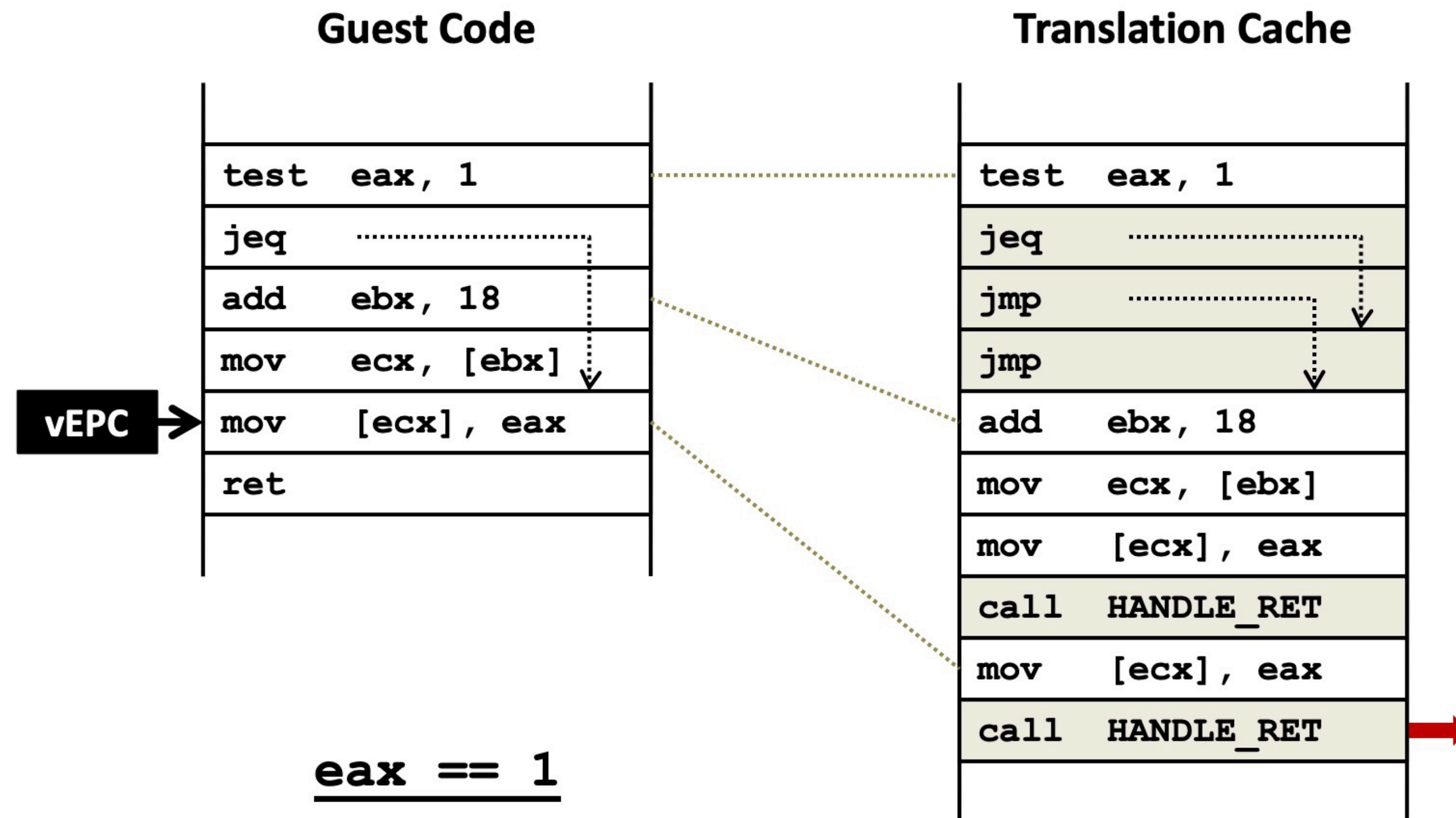
Controlling Control Flow



Controlling Control Flow



Controlling Control Flow



Virtualization Approach 4: Direct Execution with Hardware-Assisted Virtualization

- Adds a new mode so that sensitive operations could all be properly handled
- Other hardware support to make virtualization easier/faster

Hardware-Assisted CPU Virtualization (Intel VT-x)

- Two new modes of execution (orthogonal to protection rings)
 - VMX root mode: same as x86 without VT-x
 - VMX non-root mode: runs VM, sensitive instructions cause transition to root mode, even in Ring 0
- New hardware structure: VMCS (virtual machine control structure)
 - One VMCS for one virtual processor
 - Configured by VMM to determine which sensitive instructions cause VM exit
 - Specifies guest OS state

Comparison of Pre VT-x and Post VT-x

Guest Applications

Ring 3

Guest OS

Ring 1

Hypervisor

Ring 0

Hardware w/o VT-x

Guest Applications

VMX non-root
Ring 3

Guest OS

VMX non-root
Ring 0

Host Applications

VMX root Ring 3

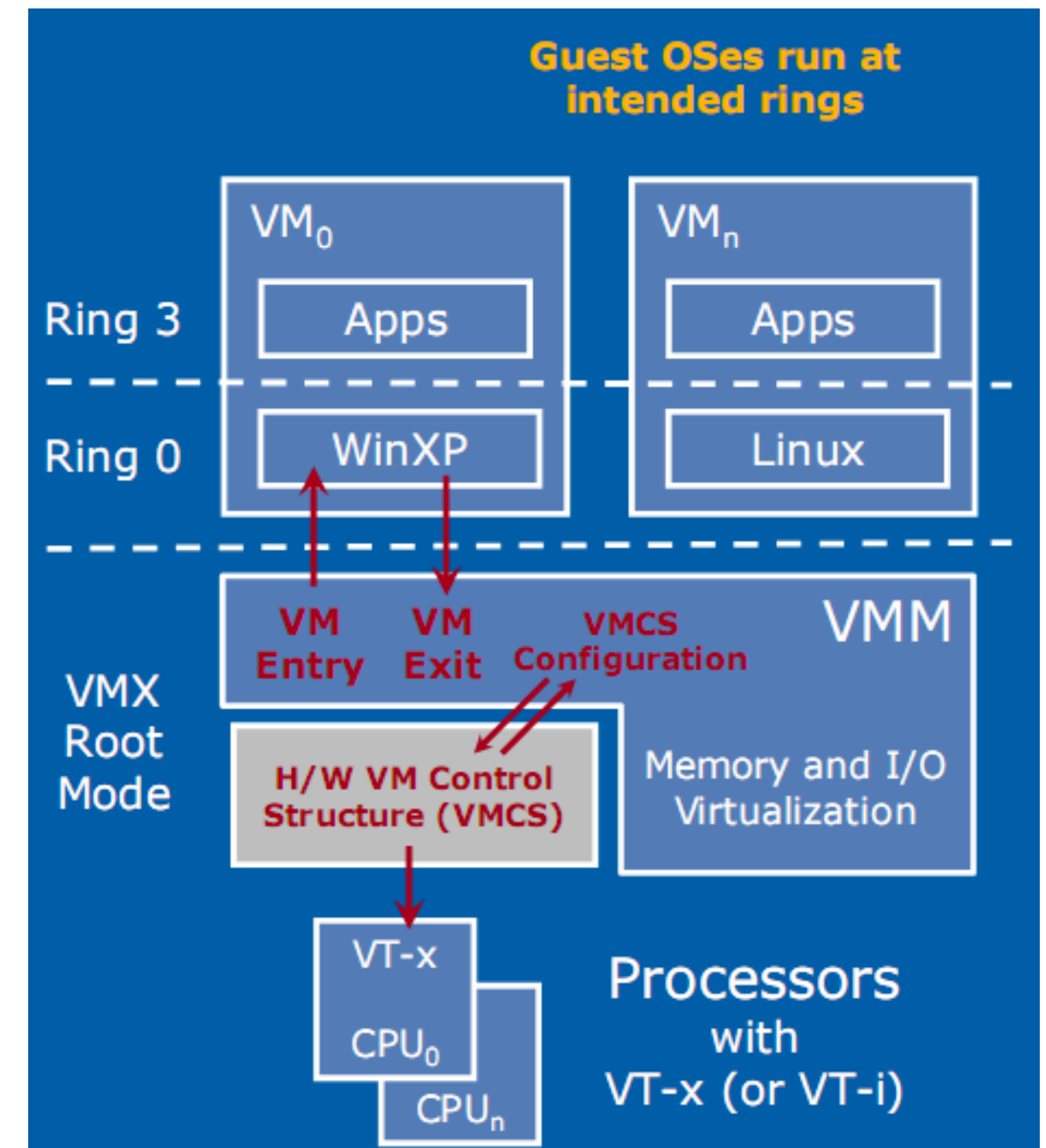
Hypervisor

VMX root Ring 0

Hardware w/ VT-x

VMX Mode Transition with Intel VT-x

- VM exit/entry (to/from root mode)
 - Registers and address space swapped in one atomic operation
 - Guest- and host-states saved and loaded to VMCS during transitions
- Whenever possible, sensitive instructions only affect states within the VMCS instead of always trapping (VM exit)
- VM exit
 - *vmcall* instruction
 - EPT page faults (more next lecture)
 - Interrupts
 - Some sensitive instructions (configured in VMCS)
- VM entry
 - *vmlaunch* instruction: enter with a new VMCS
 - *vmresume* instruction: enter for the last VMCS
- Typical vm exit/enter takes ~200 cycles on modern CPU



Example: Guest syscall with Hardware Virtualization

- VMM fills VMCS interrupt table for guest OS (including a syscall handlers)
 - and sets bit in VMCS to not exit on syscalls
- VMM executes VM entry
- Guest application invokes a syscall
- does not trap (no VMM involvement), but go to the VMCS interrupt table, which jumps to the guest OS's syscall handler

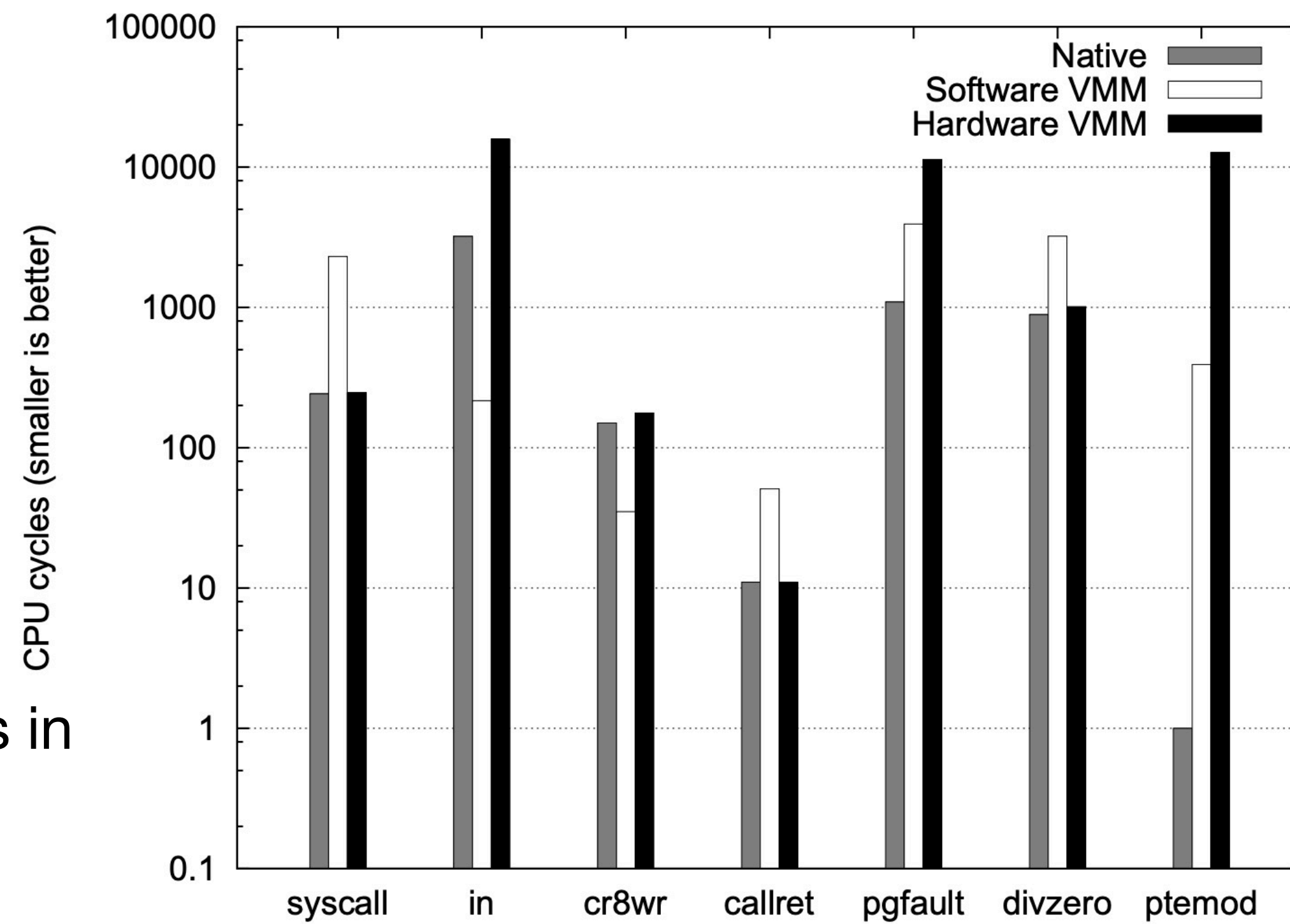
Software Binary Translation vs. Hardware-Assisted Virtualization

- Software binary translation wins in

- Trap elimination
- Emulation speed
- Callout avoidance

- Hardware-assisted virtualization wins in

- Code density
- Precise exceptions
- Syscalls



syscall, call/ret, divzero

=> native = hardware > software

in, cr8wr

=> software > native > hardware

pgfault, ptemod

=> native > software > hardware

Figure 4. Virtualization nanobenchmarks.

Virtualization Approach 5: Direct Execution with Paravirtualization

- Full virtualization (no guest OS modification)
 - Tricky and has performance overhead
- Para-virtualization: modified guest OS
 - Change (rewrite) guest OS to remove sensitive but unprivileged instructions and to use other tricks to make virtualization faster
 - Guest OS works with hypervisor (i.e., knows that it is a VM) and has some exposure to hardware
 - e.g., guest OS informs hypervisor of page table changes
 - e.g., guest OS directly calls hypervisor on system calls (*hypercalls*)
 - Guest applications are still unmodified
 - Pros and Cons?

Other Virtualization Approaches

- Container: Essentially just a group of processes with some additional features (isolated namespace, isolated resources, etc.) (e.g., Docker)
- Unikernel: LibraryOS designed for a single application, running on hypervisor (as a VM) or host OS (as a process)
- Sandboxing: Limit what the applications (and libOS) can do (e.g., gVisor)
- Language-based: Running applications written in a high-level language on language runtimes (e.g., JVM)

Virtualization Approaches Summary

- Hosted interpretation
 - Interpret each instruction, super slow (e.g., Virtual PC on Mac)
- Direct execution with trap-and-emulate
 - Requires a virtualizable processor and only works for the same architecture
- Direct execution with binary translation
 - Works with non-virtualizable processor, but implementing VMM is tricky
- Direct execution with hardware-assisted virtualization
 - Needs new generation of hardware (which is the norm now), mode switching is still not optimized
- Direct execution with paravirtualization
 - Good performance and works with non-virtualizable processors, but require guest OS changes
- OS-level virtualization, library-level, language (app)-level, unikernels, etc.
 - More lightweight and faster to start, but less secure