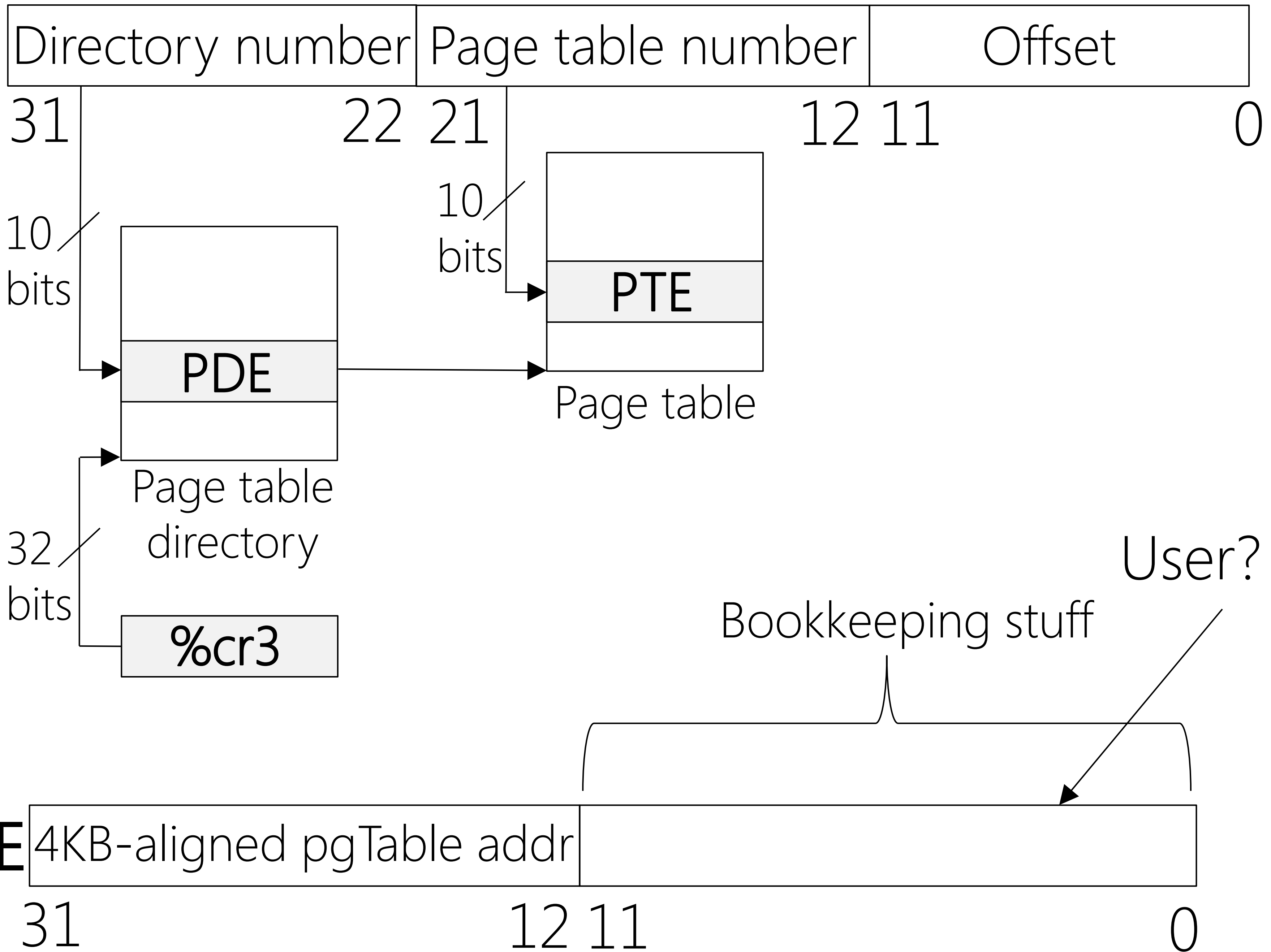
Virtual Memory, Part II

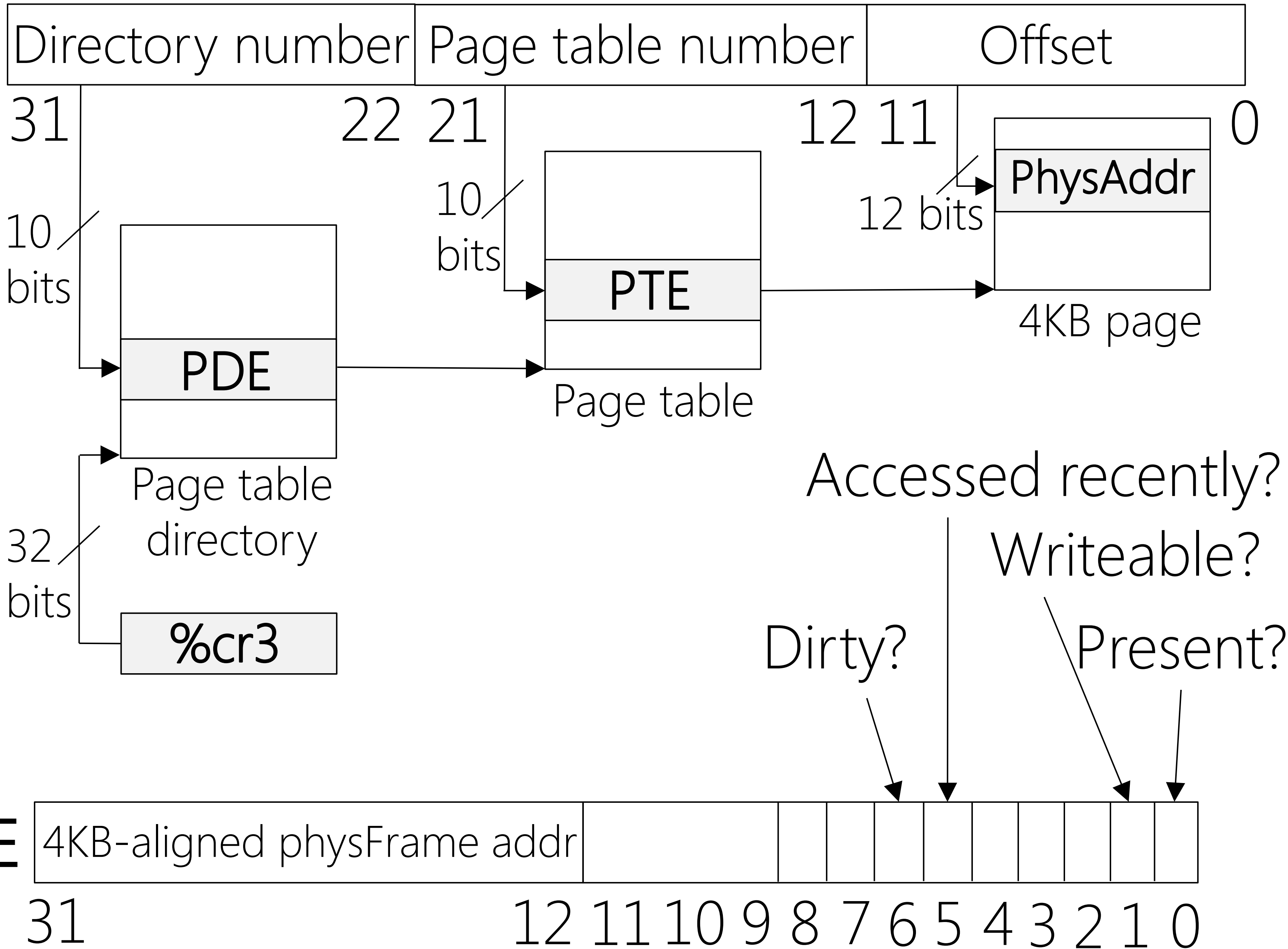
Goals of Virtual Memory

- Allow physical memory to be smaller than virtual memory—applications receive illusion of huge address spaces!
  - At any given time, a process' virtual address space may be fully in RAM, partially in RAM, or not in RAM at all
  - Spare applications the chore of moving pages between memory and disk
- Provide memory isolation between processes and the OS memory (but allow sharing when desired!)
- How do systems implement paging in real life?

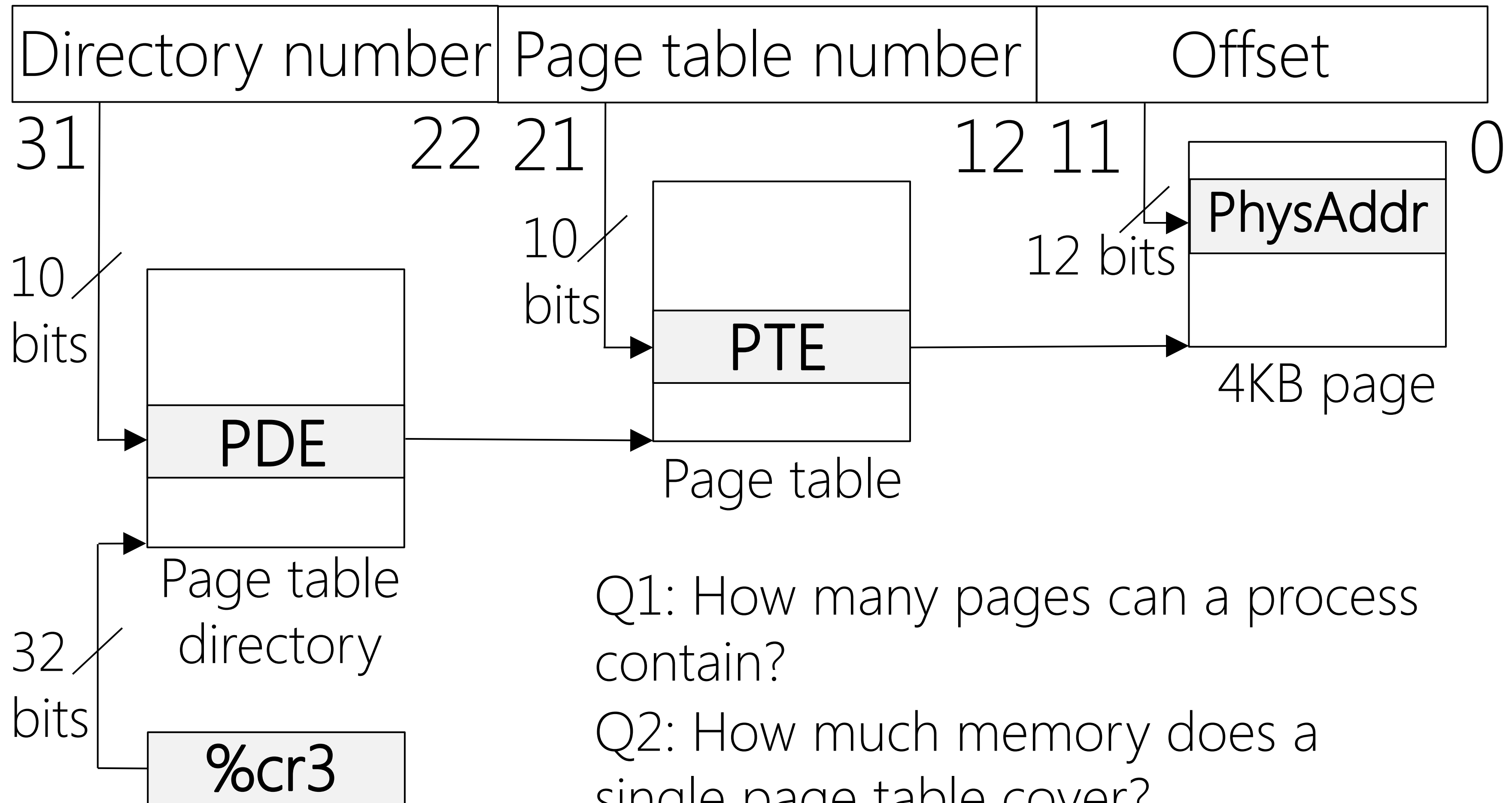
Case study: x86



Case study: x86



Case study: x86



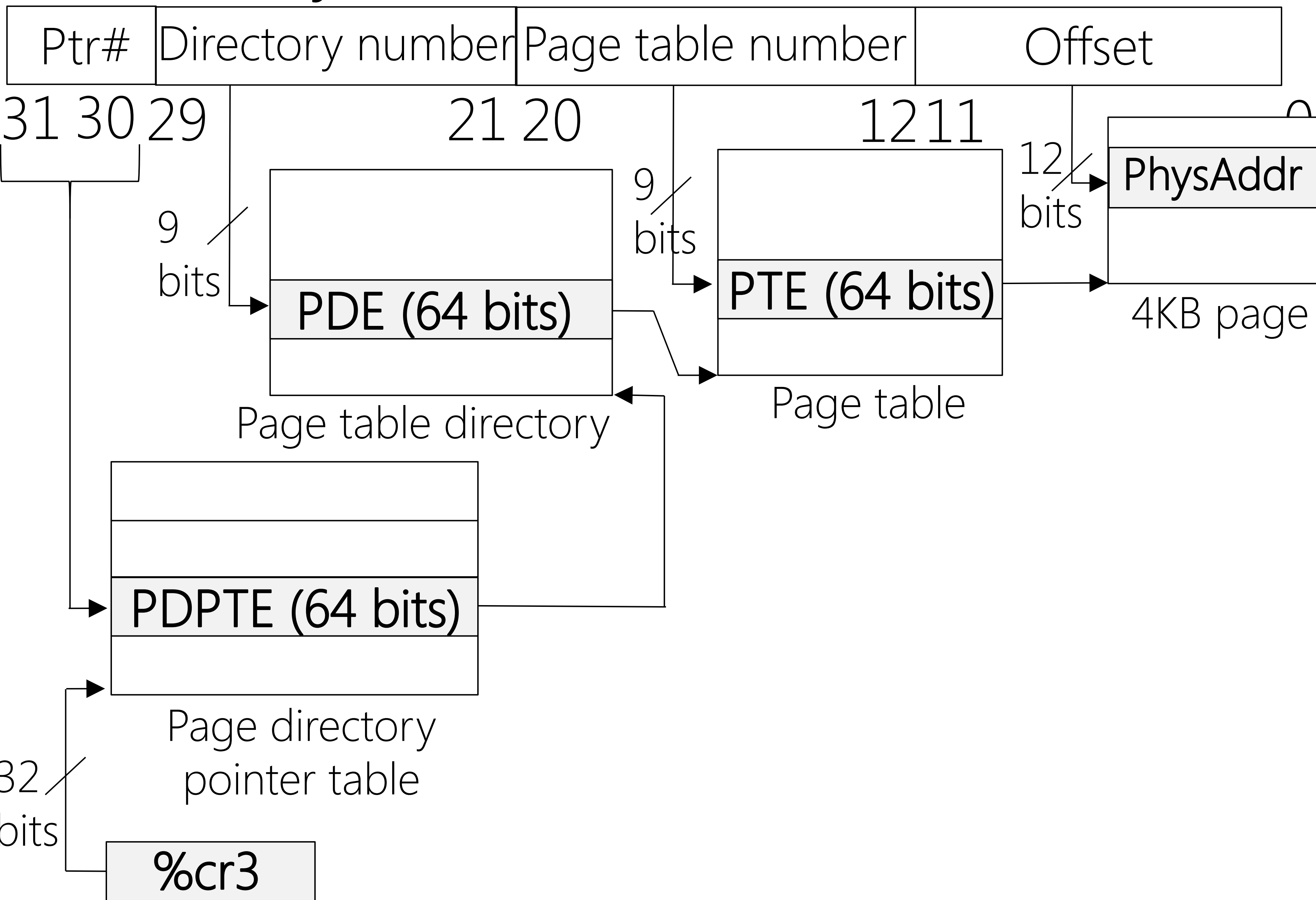
Q1: How many pages can a process contain?

Q2: How much memory does a single page table cover?

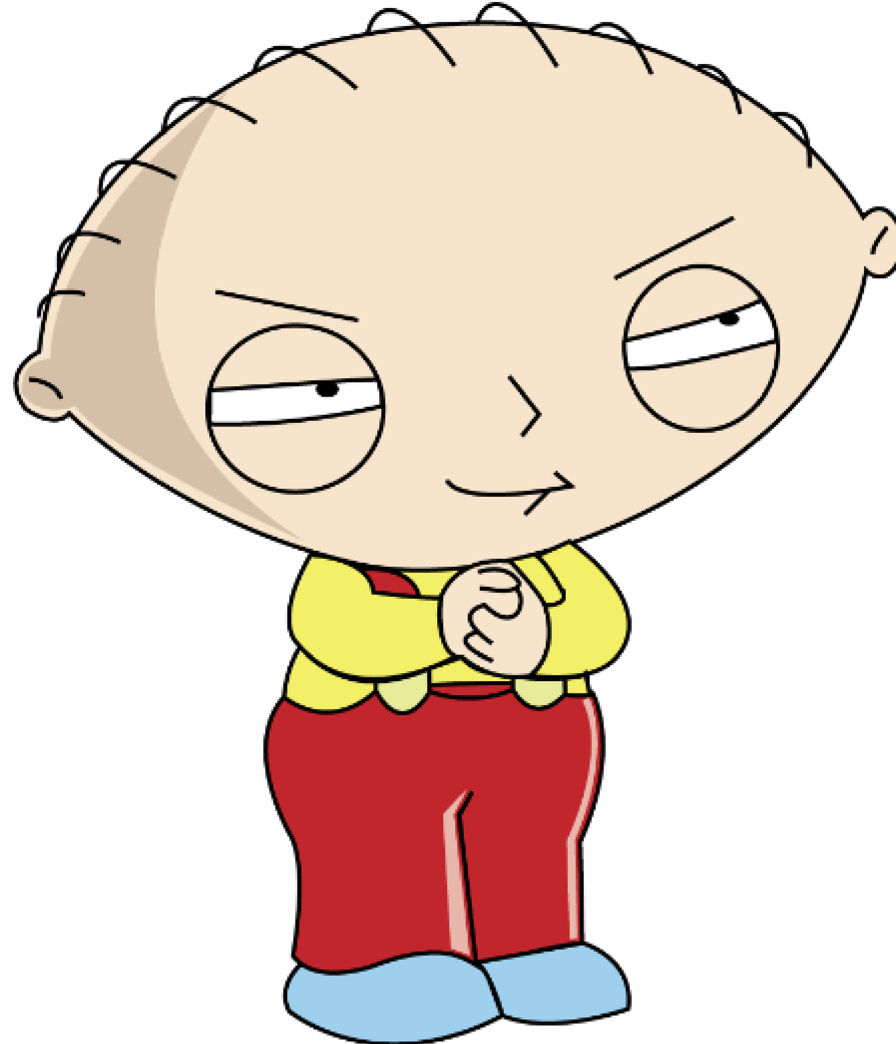
Q3: What is the minimum size of a machine's physical memory?

Q4: What is the maximum size of a machine's physical memory?

x86 Physical Address Extension (PAE)



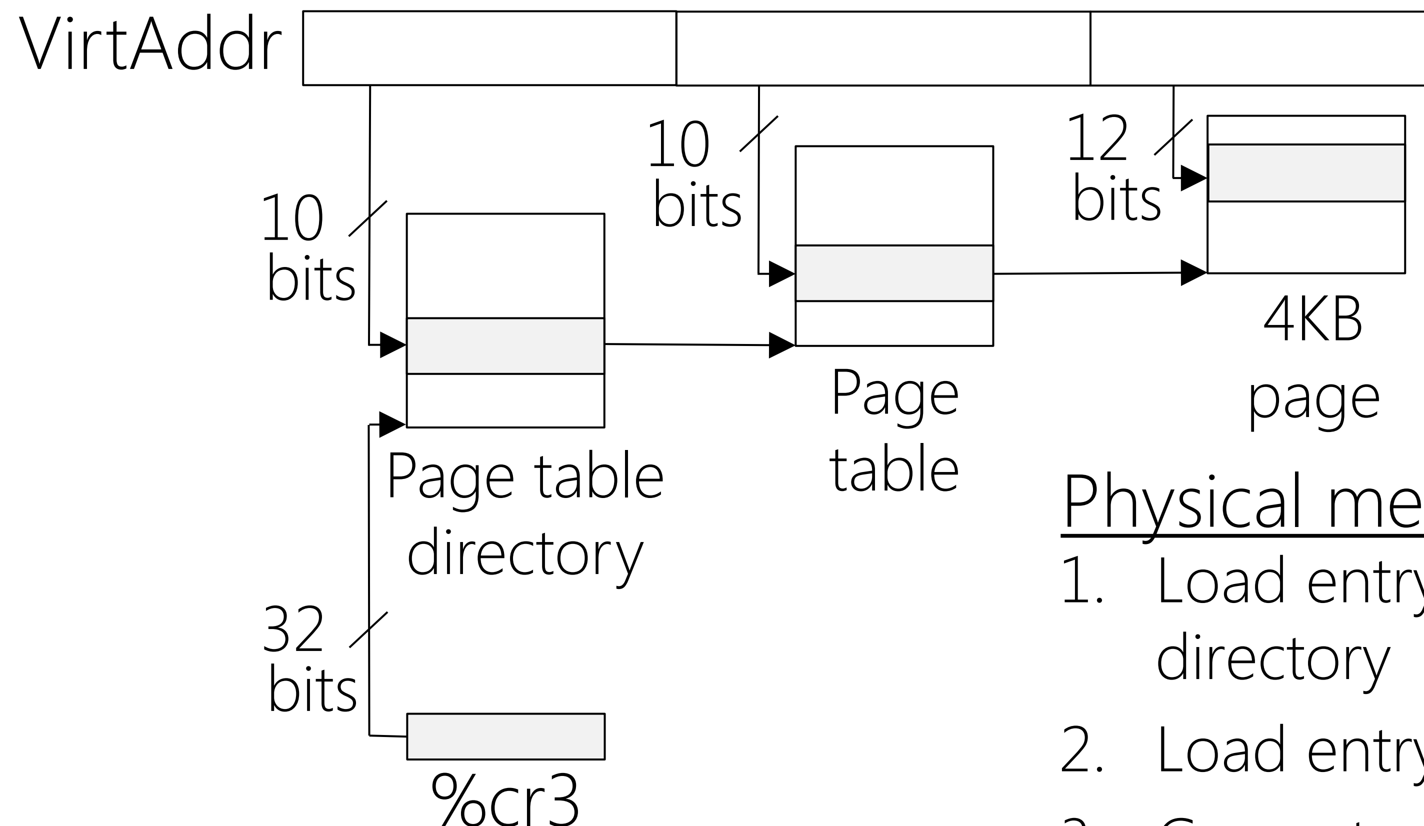
Q: What Do Page Tables Look Like On MIPS R3000?



A: You Get To Decide!

Paging: The Good and the Bad

- Good: A virtual address space can be bigger than physical memory
- Bad: Each virtual memory access now requires at least two physical memory accesses

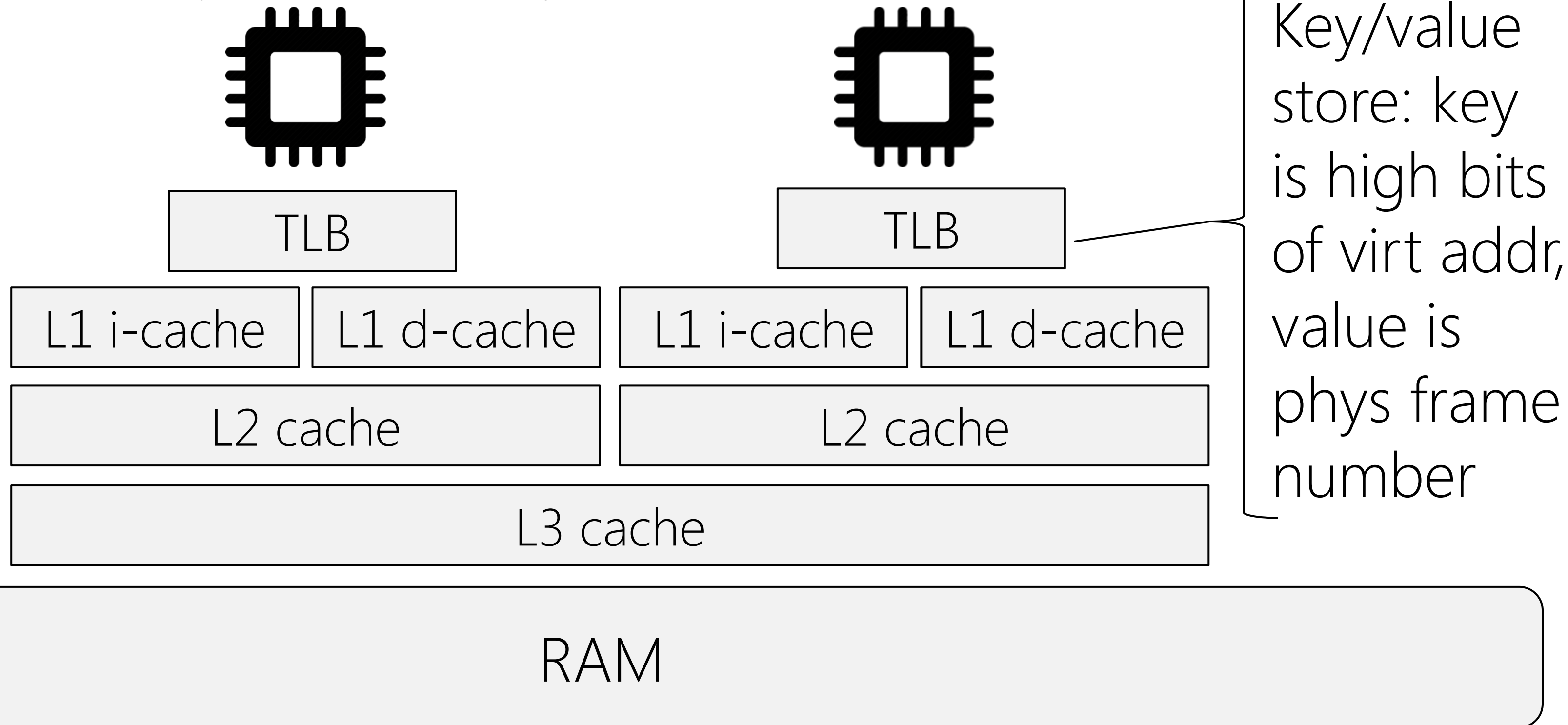


<u>Physical memory accesses</u>

1. Load entry from page table directory
2. Load entry from page table
3. Generate the “real” memory access

Translation Lookaside Buffers (TLBs)

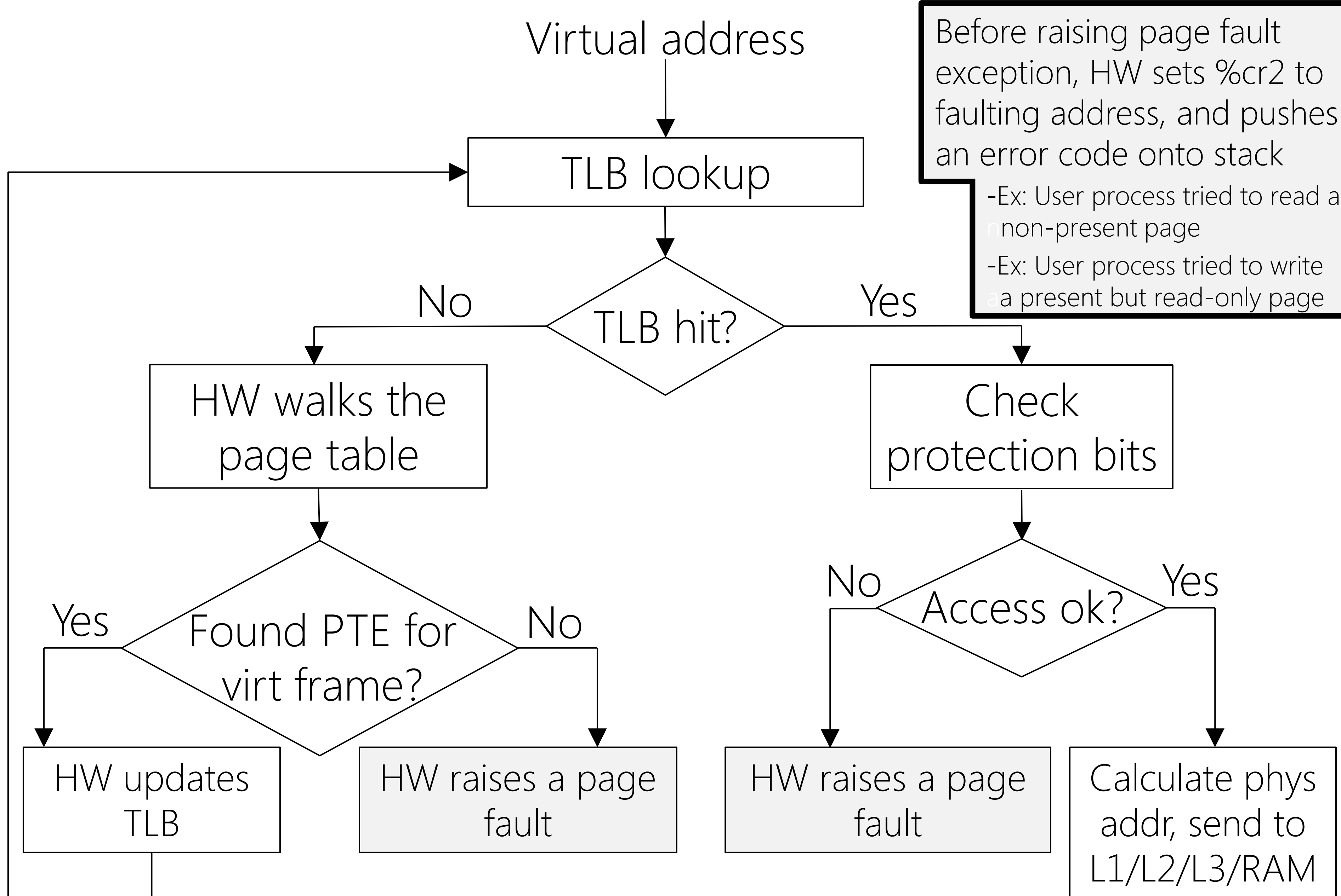
- Idea: Cache some PTEs in small hardware buffer
  - If virtual address has an entry in TLB, don't need to go to physical memory to fetch PTEs!
  - If virtual address misses in TLB, we must pay at least one physical memory access to fetch PTE



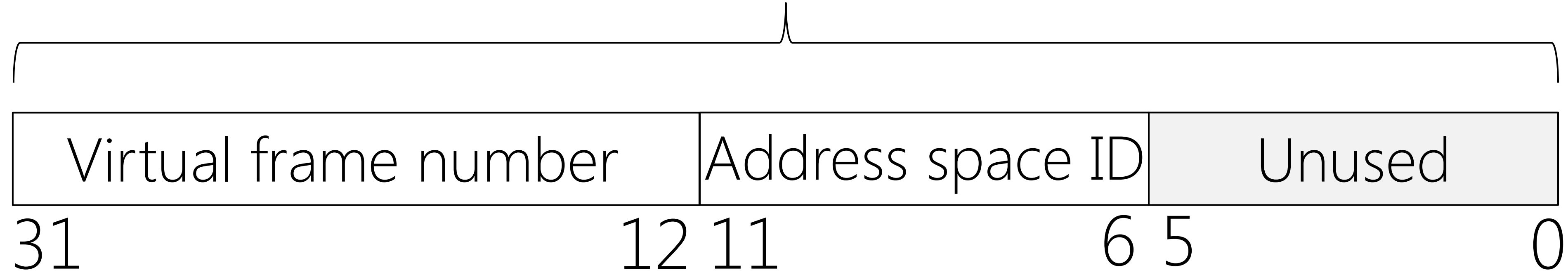
Translation Lookaside Buffers (TLBs)

- Idea: Cache some PTEs in small hardware buffer
  - If virtual address has an entry in TLB, don't need to go to physical memory to fetch PTEs!
  - If virtual address misses in TLB, we must pay at least one physical memory access to fetch PTE
- TLBs are effective because programs exhibit locality
  - Temporal locality: When a process accesses virtual address x, it will likely access x again in the future (Ex: a function's local variable that lives on the stack)
  - Spatial locality: When the process accesses something at memory location x, the process will likely access other memory locations close to x (Ex: reading elements from an array on the heap)

The Lifecycle of a Memory Reference on x86



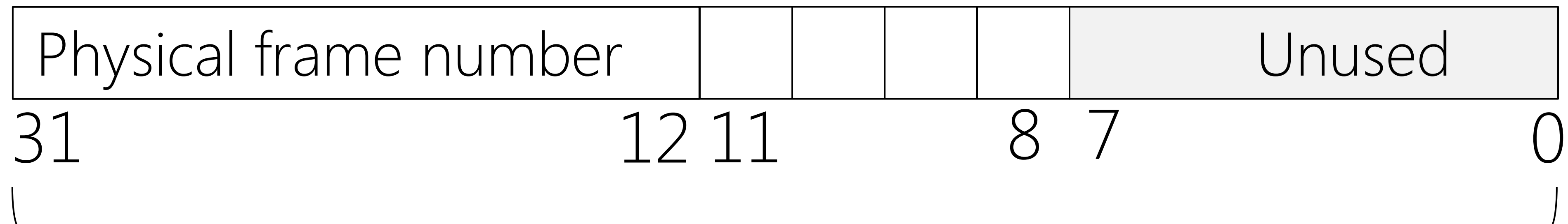
MIPS R3000: Interacting with the TLB



Writable?

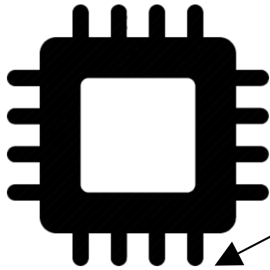
Valid?

Global?

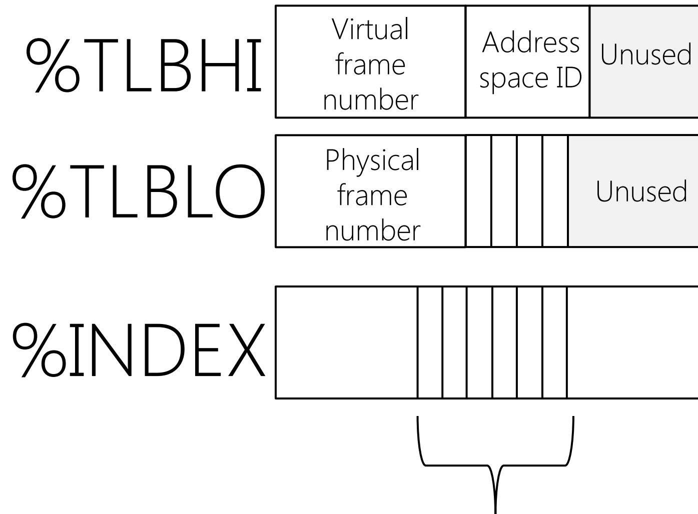


A single TLB entry:
a TLBHI structure +
a TLBLO structure

MIPS R3000: Interacting with the TLB

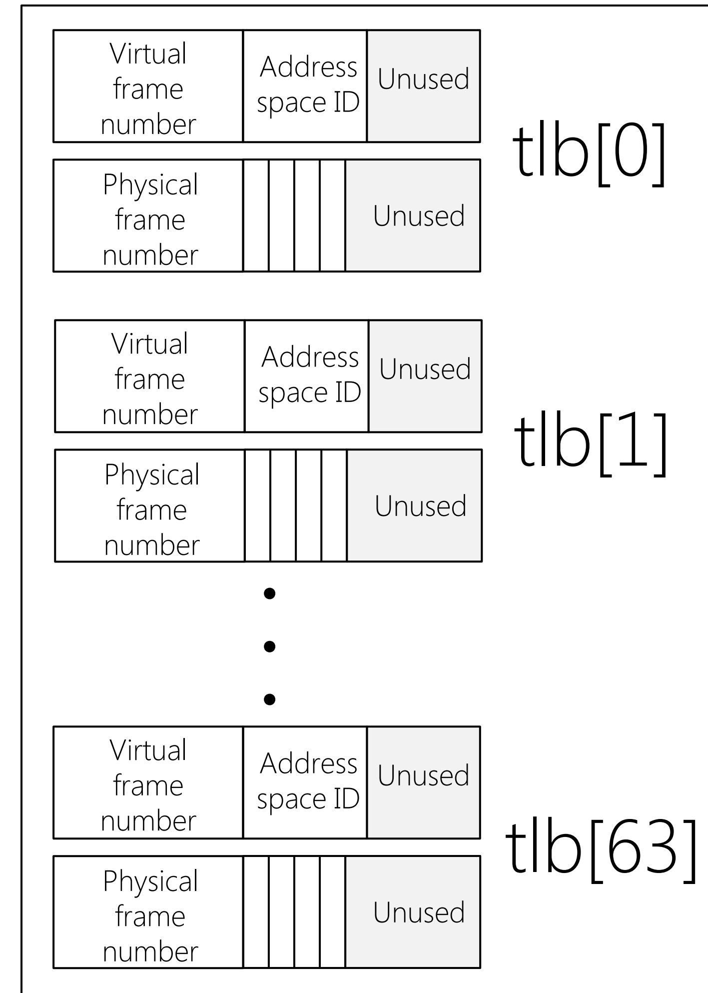


OS must set ASID during context switch!

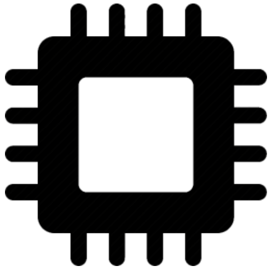


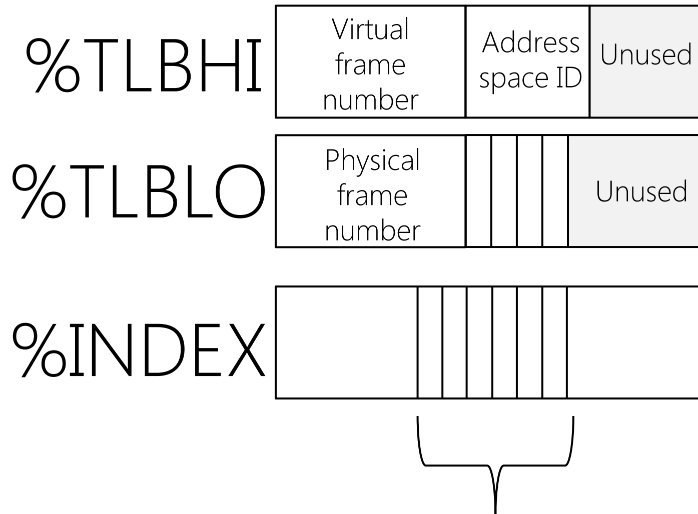
Used to select tlb[] entry for:

- TLBR: Read TLB entry into %TLBHI and %TLBLO
- TLBWI: Write %TLBHI and %TLBLO to TLB entry



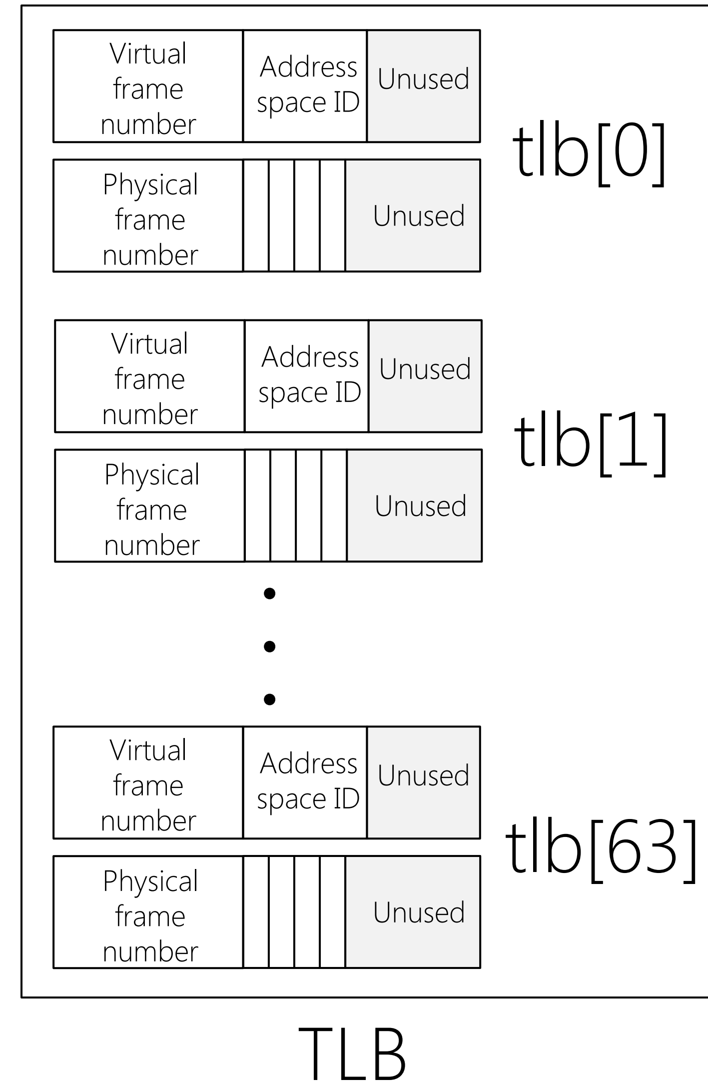
MIPS R3000: Interacting with the TLB



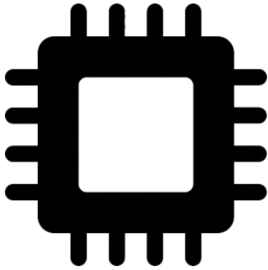


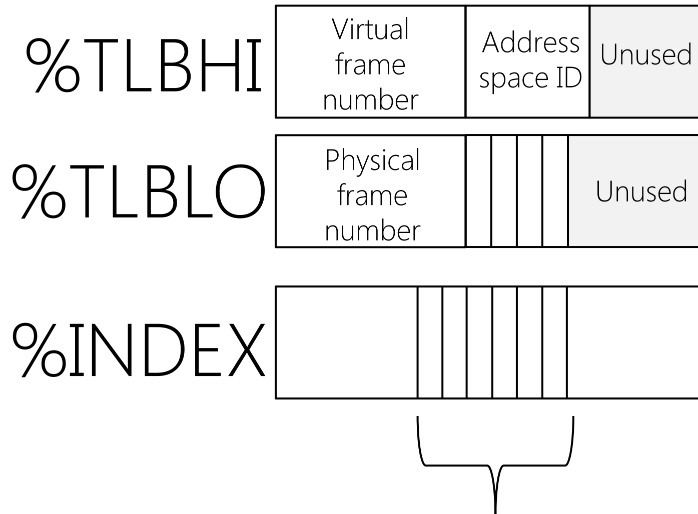
Set to tlb[] index by:

- TLBP: Search TLB for entry matching %TLBHI, set INDEX to matching TLB entry or -1 if no match found



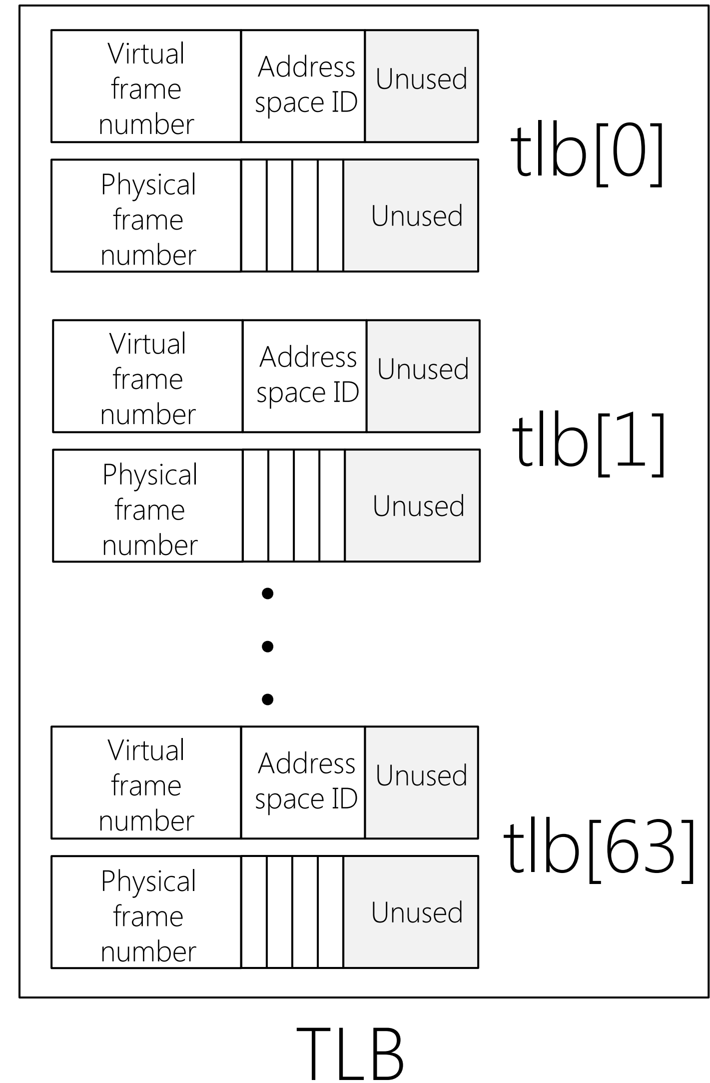
MIPS R3000: Interacting with the TLB



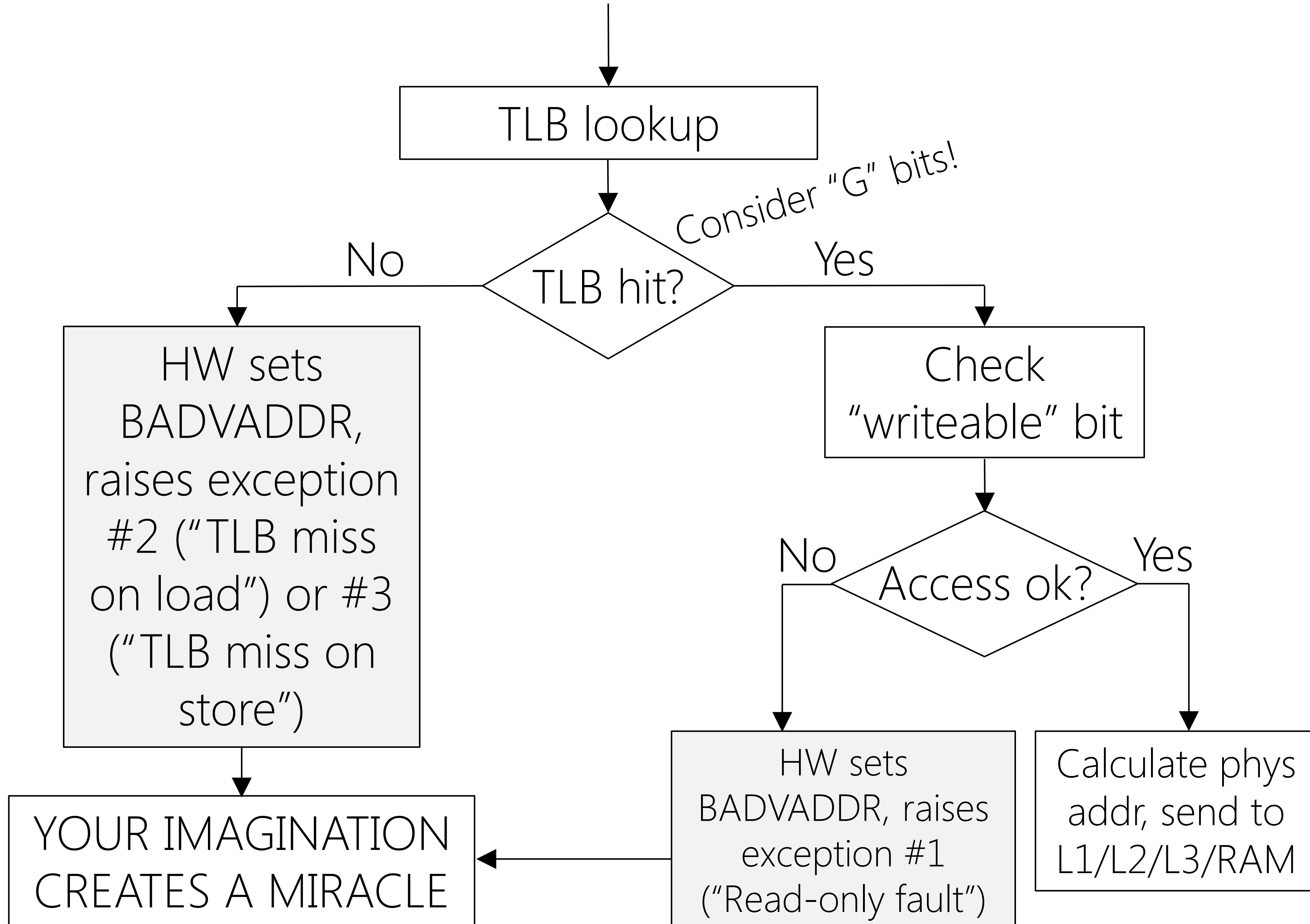


Not used by:

- TLBWR: Write %TLBHI and %TLBLO to random TLB entry



The Lifecycle of a Memory Reference on MIPS



TLBs and Context Switches

- If TLB entries are tagged with ASIDs:
  - OS updates current ASID (e.g., by setting TLBHI::ASID on MIPS)
  - OS doesn't need to flush TLBs
  - Even if OS occasionally has to evict entries, this is better than having to evict ALL entries during EVERY context switch (since this generally requires size(TLB) page table walks when a new task starts to warm TLB)
  - Scheduler can reduce invalidations with AS-to-core affinity
- If TLB entries are *not* tagged with ASIDs:
  - OS must invalidate all TLB entries during a context switch
  - x86: Writing to %cr3 on x86—this updates PDE pointer and invalidates all TLB entries
  - MIPS: OS can use constant value for all ASIDs, and manually invalidate all TLB entries during context switch

TLB Invalidations

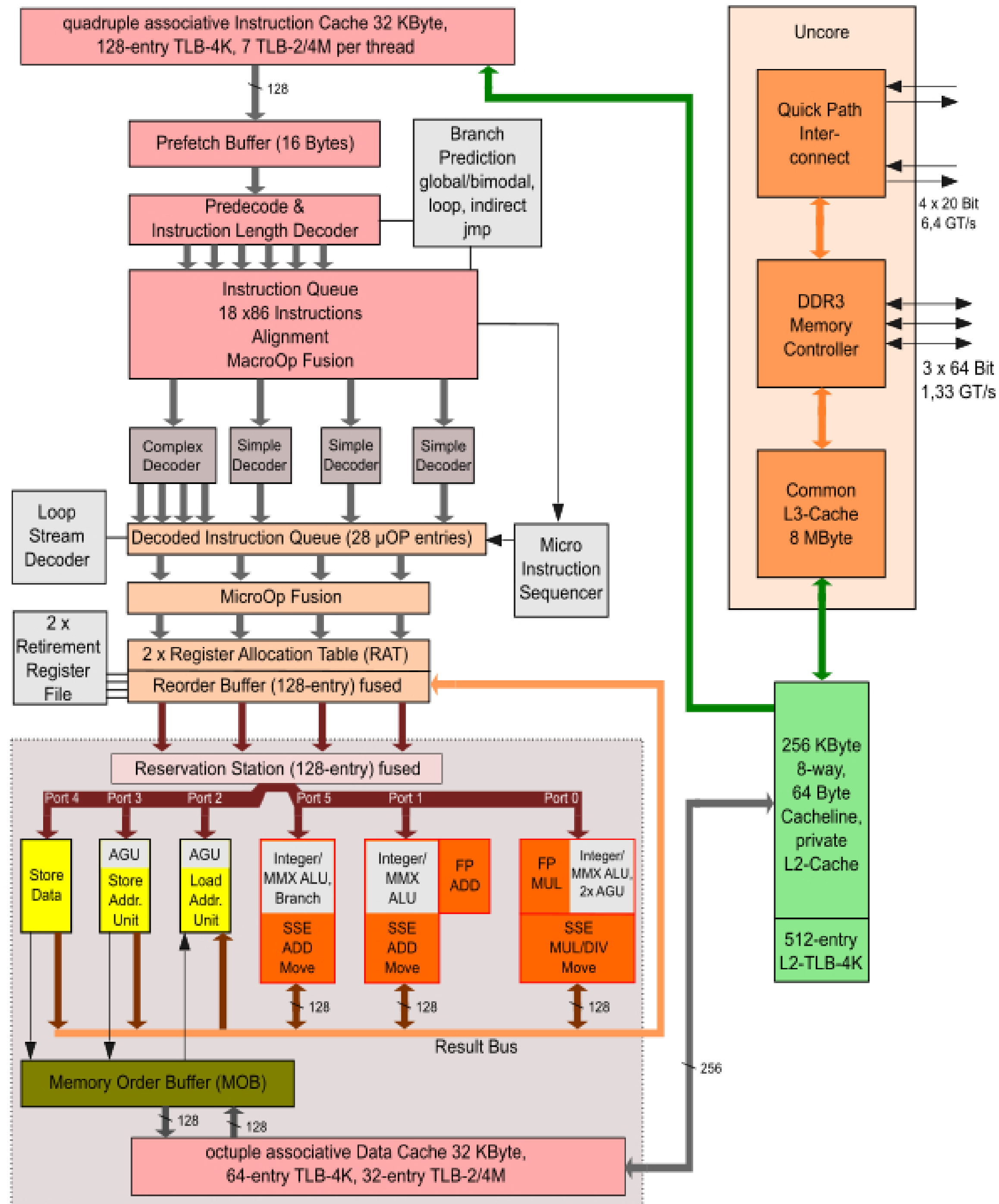
- When OS changes a PTE, must also invalidate any matching TLB entry!
  - x86: “INVLPG virtAddr” invalidates individual TLB entry
  - MIPS: Use “TLBP” (the TLB probe instruction) to set %INDEX to that of the TLB entry to invalidate; then, use “TLBWI” to overwrite it
- On a multicore machine, PTEs from a single address space can be mapped into multiple per-core TLBs
  - If a core wants to modify a PTE entry, it must send cross-core interrupts to other cores
  - Once other cores are spin-waiting, first core modifies PTE then wakes up other cores
  - Other cores invalidate relevant TLB entries and resume execution

TLB Design Trade-offs

- Software-managed TLB
  - Good: OS has freedom to design page tables, page directories, and other arbitrarily interesting structures
  - Good: OS has freedom to design TLB eviction policy that might be too complex to implement in hardware
  - Bad: Performance overhead
    - Software is slower than hardware
    - OS lacks access to low-level hardware state, so handling TLB misses in software may require discarding work that's already in the CPU pipeline

Intel Nehalem architecture

(~2013: Xeon, Core i7)

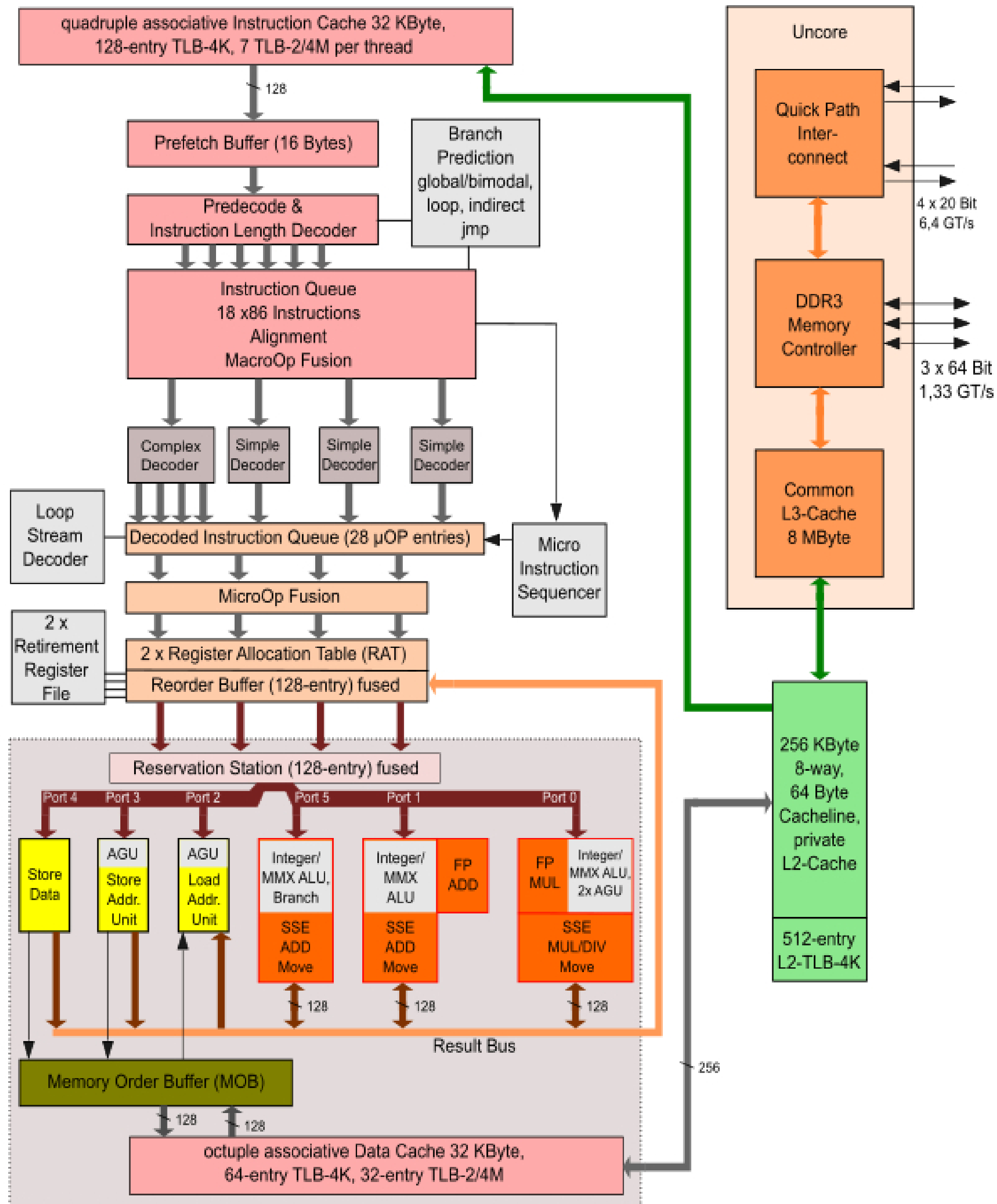


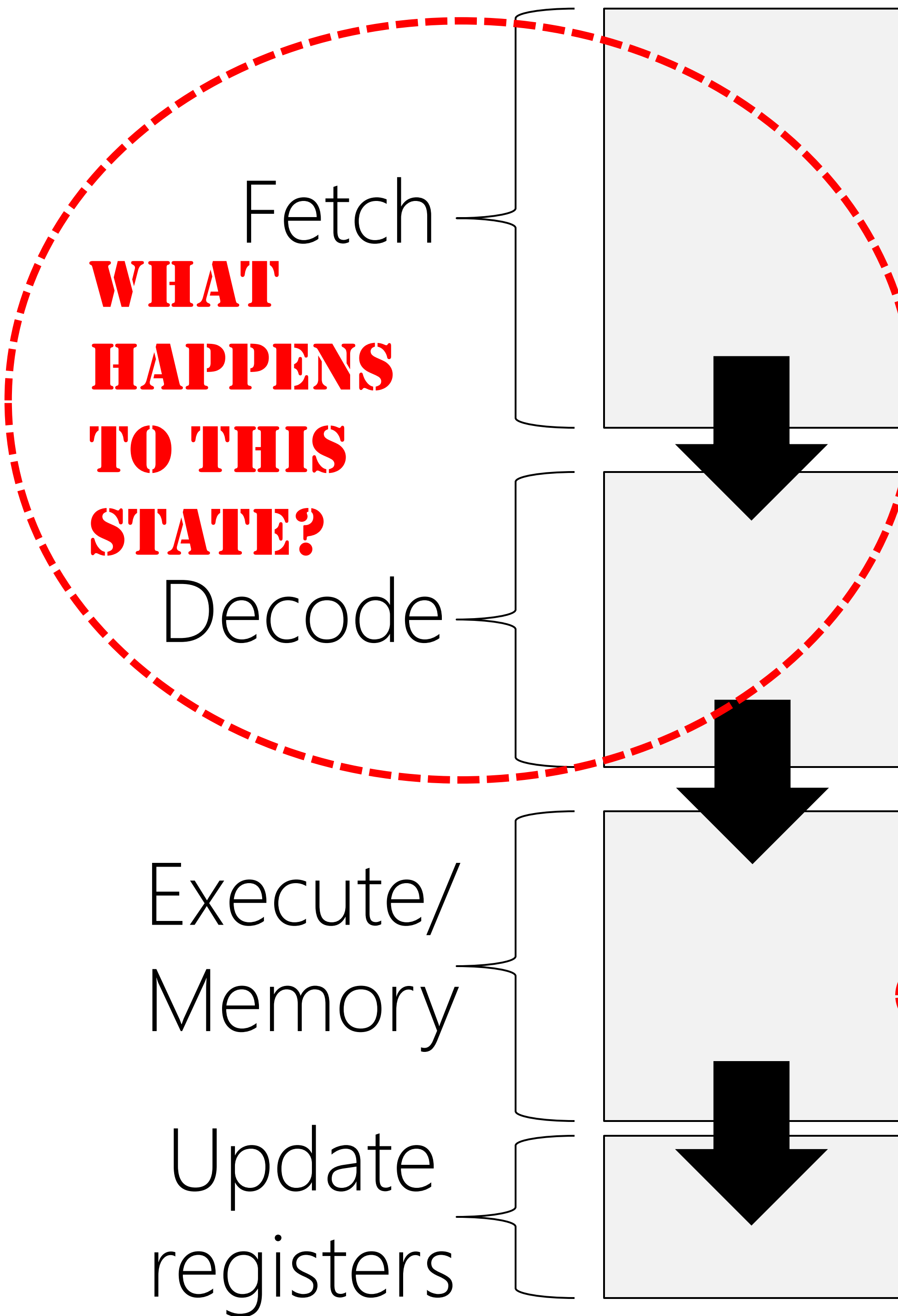
Fetch

Decode

Execute/
Memory

Update
registers






```
mov %eax, [%esp]
add %eax, 42
sub %edi, %eax
...
```

sub %edi, %eax

add %eax, 42

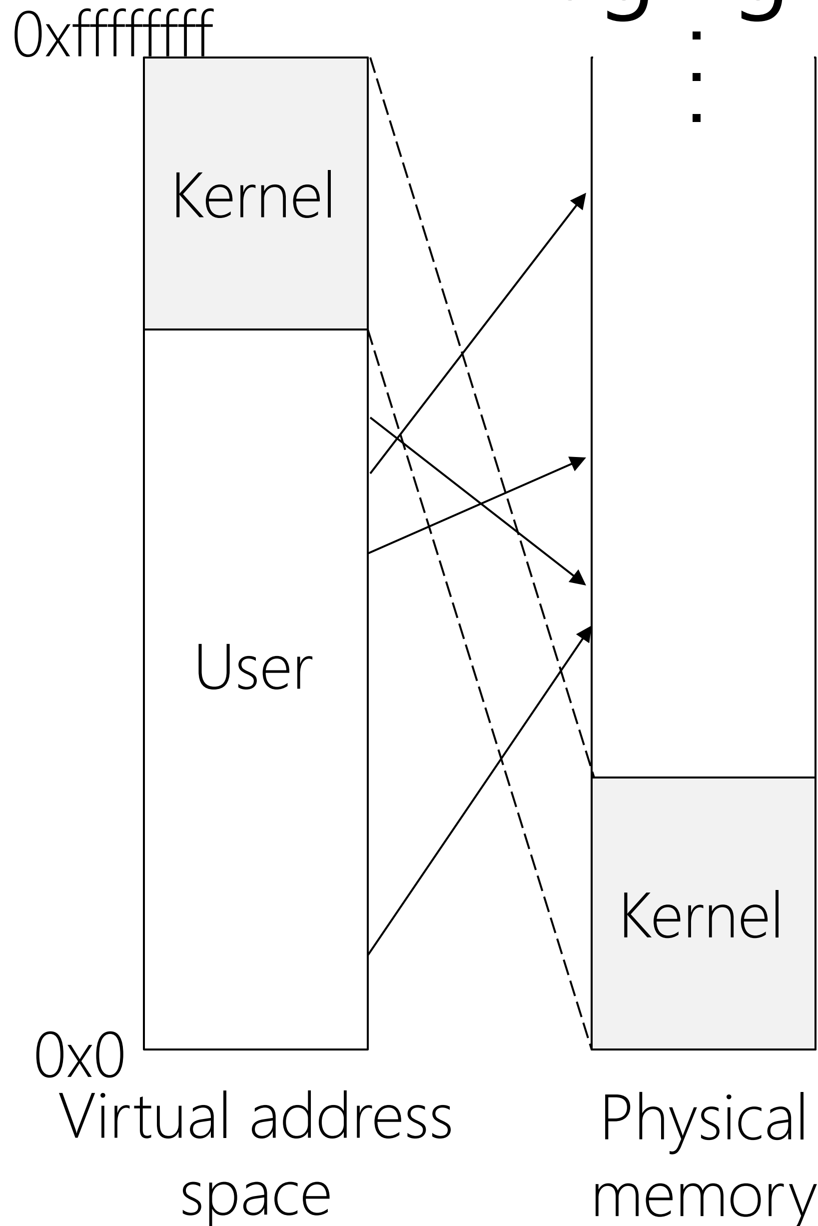
mov %eax, [%esp]

TLB MISS

TLB Design Trade-offs

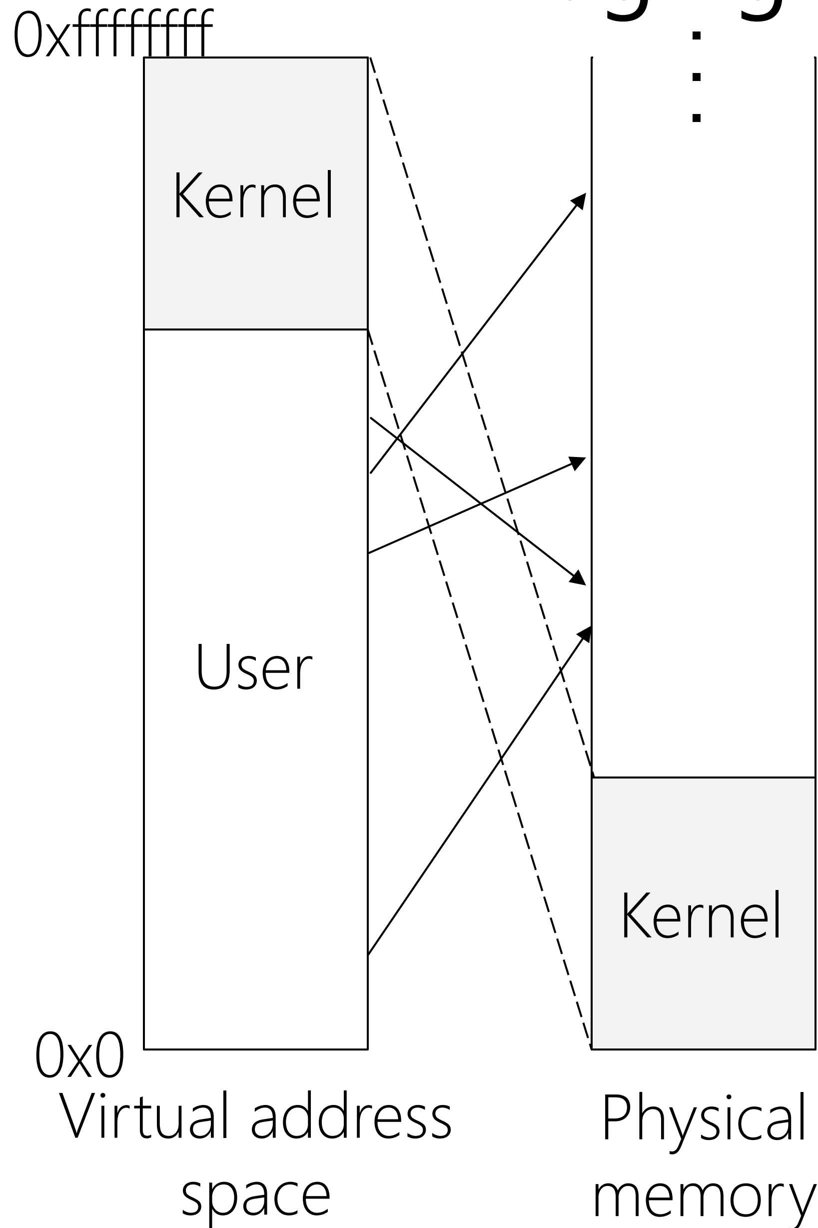
- Software-managed TLB
  - Good: OS has freedom to design page tables, page directories, and other arbitrarily interesting structures
  - Good: OS has freedom to design TLB eviction policy that might be too complex to implement in hardware
  - Bad: Performance overhead
    - Software is slower than hardware
    - OS lacks access to low-level hardware state, so handling TLB misses in software may require discarding work that’s already in the CPU pipeline
- Hardware-managed TLB
  - Good: TLB miss doesn’t cause exception that must be handled by OS
    - Hardware can just stall the current instruction . . .
    - . . . and let other instructions proceed!
  - Bad: Page table/page directory/etc can’t be changed by OS

Kernel Memory Interactions with Paging and TLB: Linux



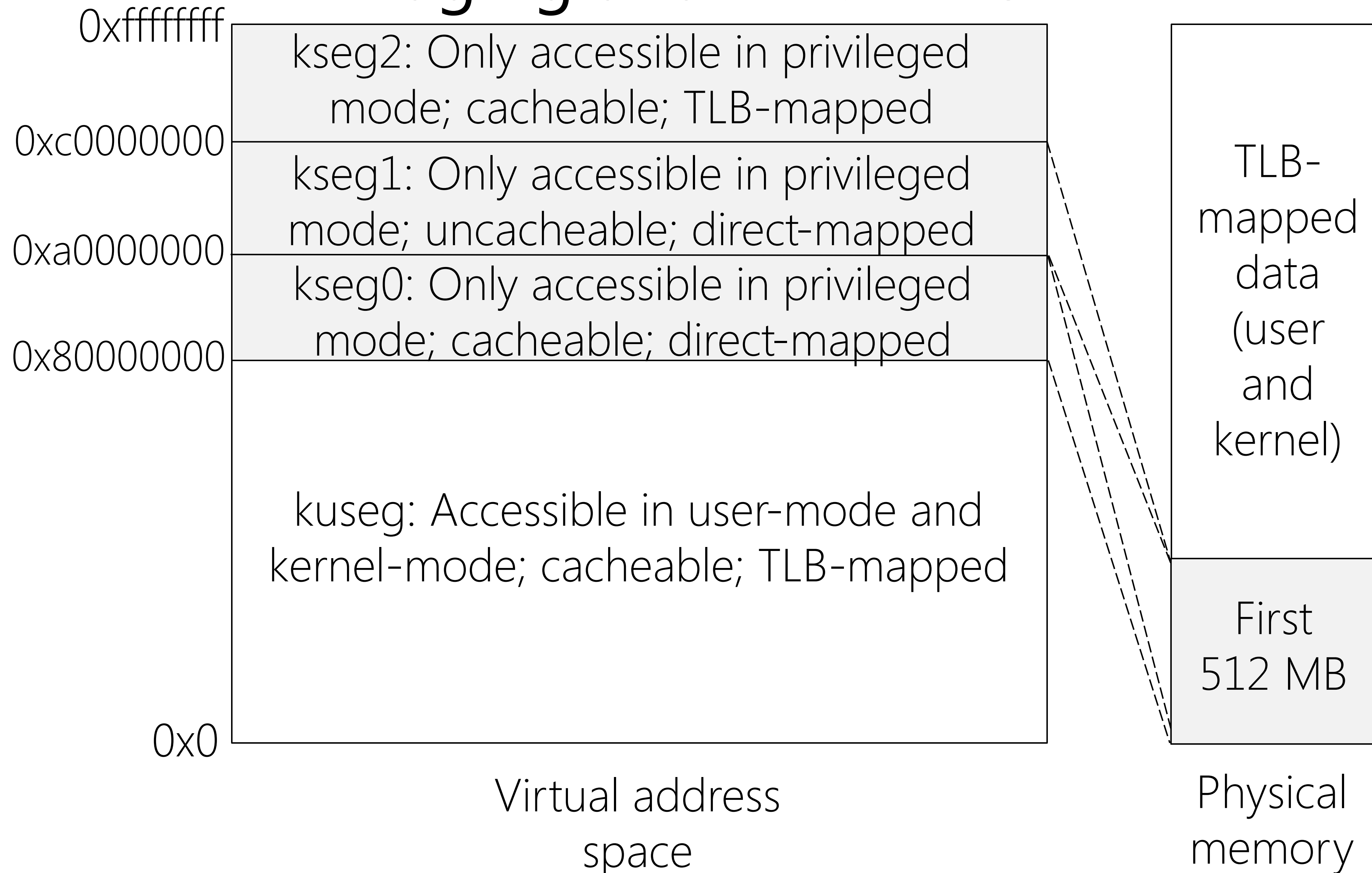
- Kernel places itself in lowest part of physical memory, but maps itself into highest part of each process' virtual address space
- Bottom part of process' address range:
  - 0x00000000—0xbfffffff are accessible by both user-mode and kernel-mode
  - Address range covered by first 768/1024 of the process' page directory; represents unique, per-process pages
- Top part of process' address range:
  - 0xc0000000—0xffffffff can only be accessed in kernel-mode
  - Address range covered by last 256 entries of page directory; "user?" bit is 0 in PDEs, so user-level code can't access those memory addresses!

Kernel Memory Interactions with Paging and TLB: Linux



- All virtual memory addresses (include ones to kernel memory) go through TLB
  - Reference to kernel memory can cause TLB miss, but not page fault—kernel is always resident in RAM!
- A system call or interrupt doesn't require changing %cr3—kernel is mapped into address space already, but now the CPU runs in privileged mode, so high virtual memory addresses can be accessed!

Kernel Memory Interactions with Paging and TLB: MIPS



"The use of one memory management organization over another has not catapulted any architecture to the top of the performance ladder, nor has the lack of any memory management function been the leading cause of an architecture's downfall. So, while it may seem refreshing to have so many choices of VM interface, the diversity serves little purpose other than to impede the porting of system software."

B. Jacob and T. Mudge, "Virtual Memory in Contemporary Microprocessors." In *IEEE Micro*, Volume 18, Issue 4, July 1998.

Important job of OS: Hide processor quirks using abstraction!