Virtual Memory, Part II

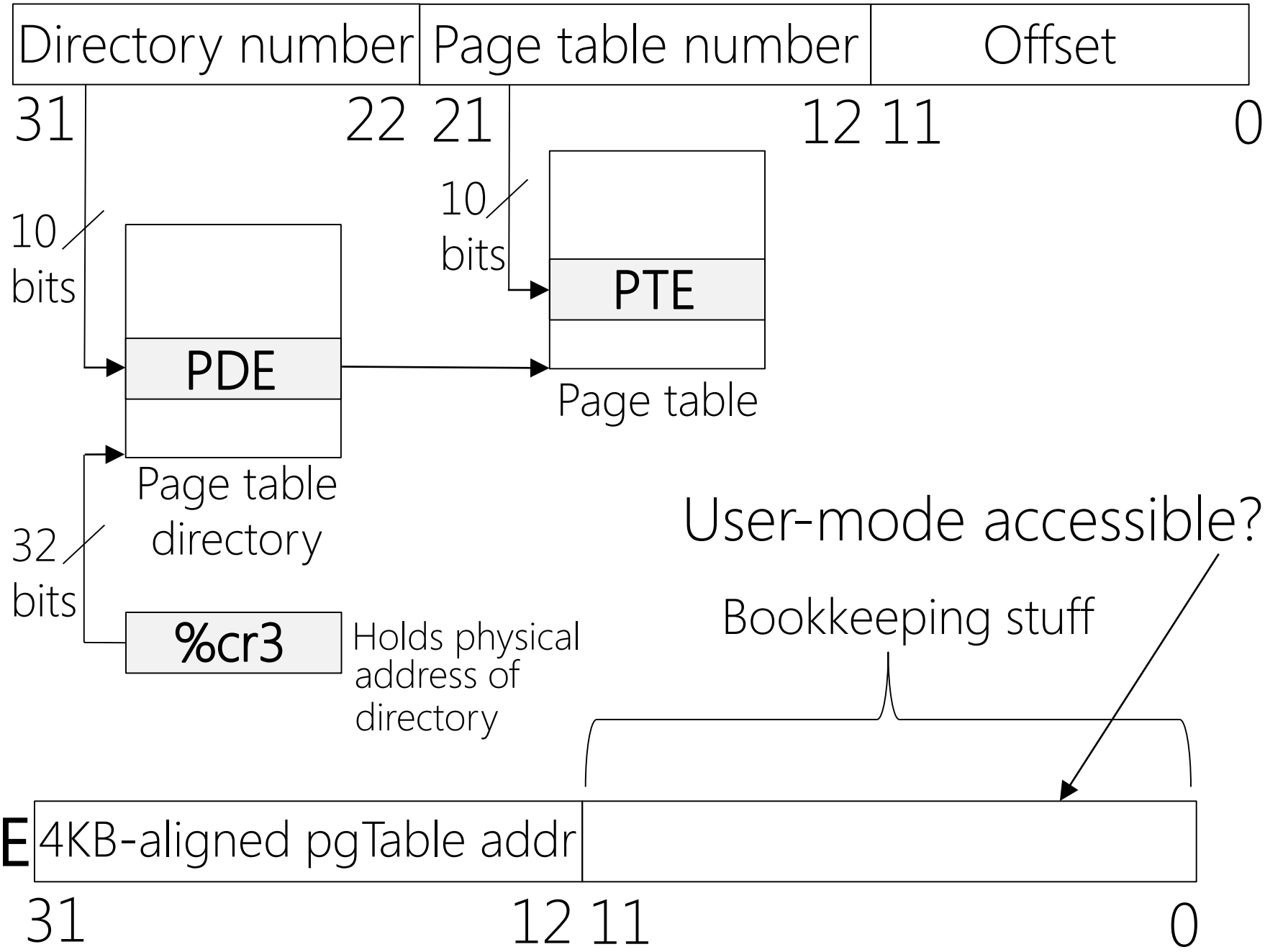
CS 161: Lecture 7

2/21/17

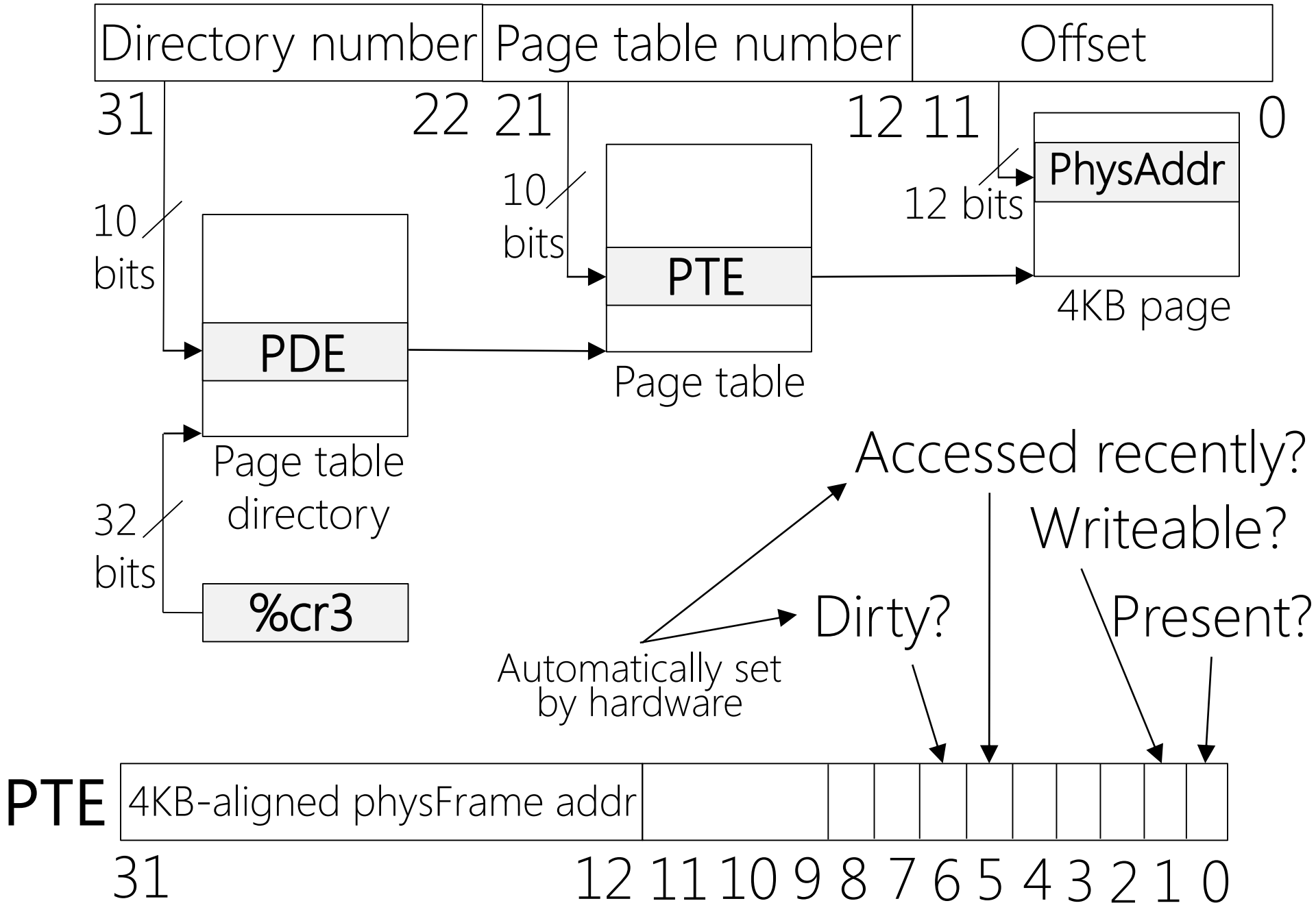
Goals of Virtual Memory

- 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
  - Automate 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?

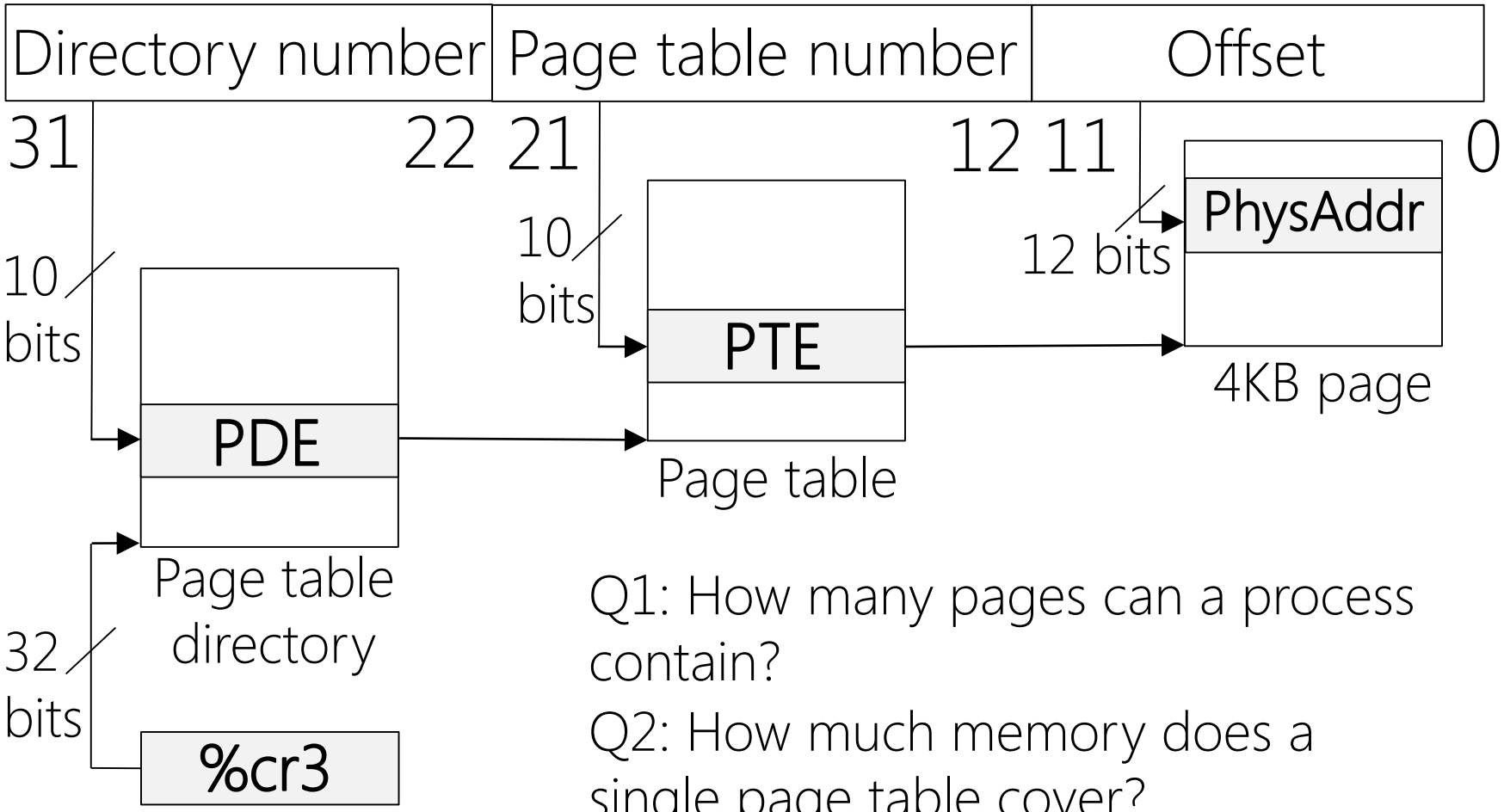
Case study: x86 (Hardware-defined page tables)



Case study: x86 (Hardware-defined page tables)



Case study: x86 (Hardware-defined page tables)



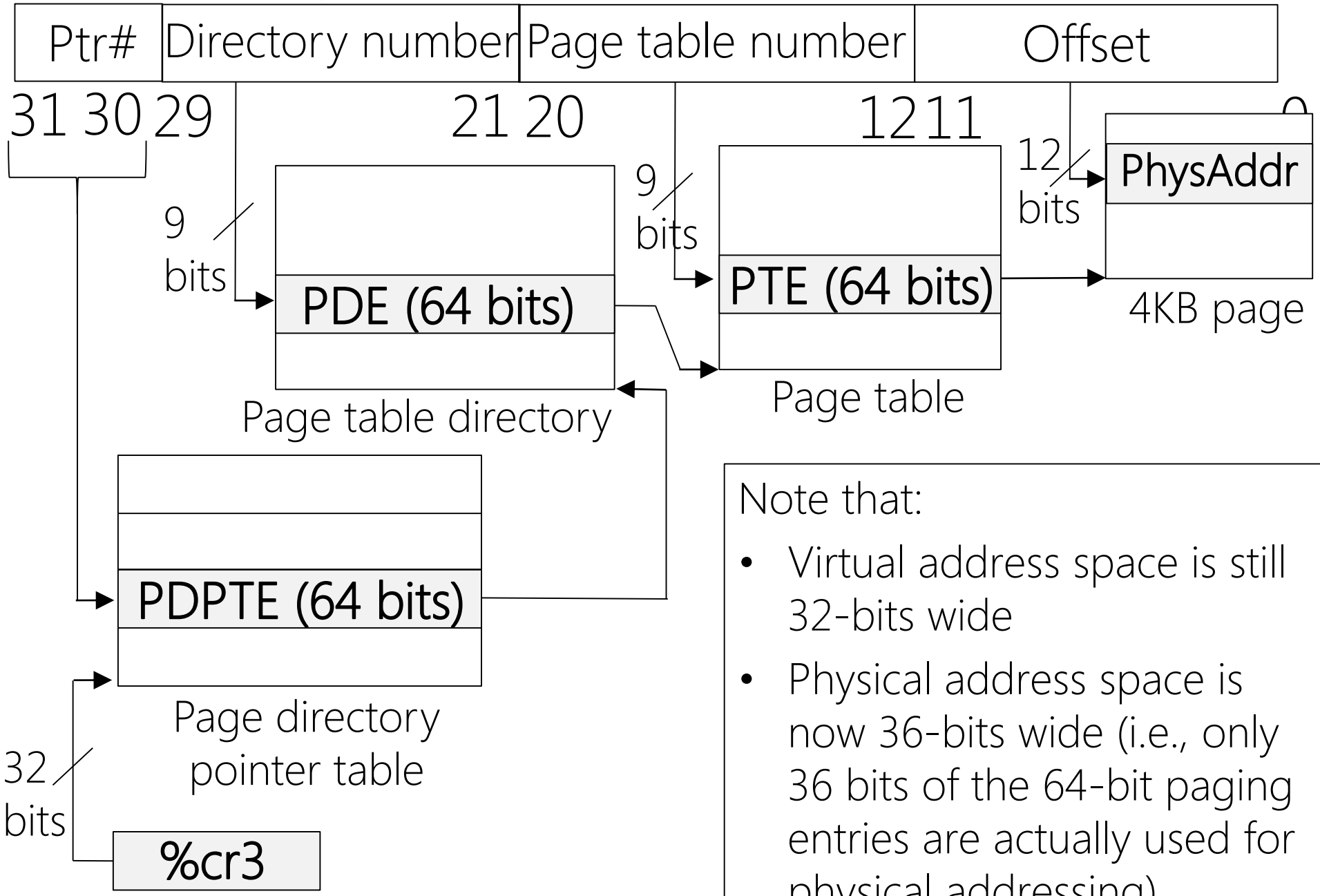
Q1: How many pages can a process contain?

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)

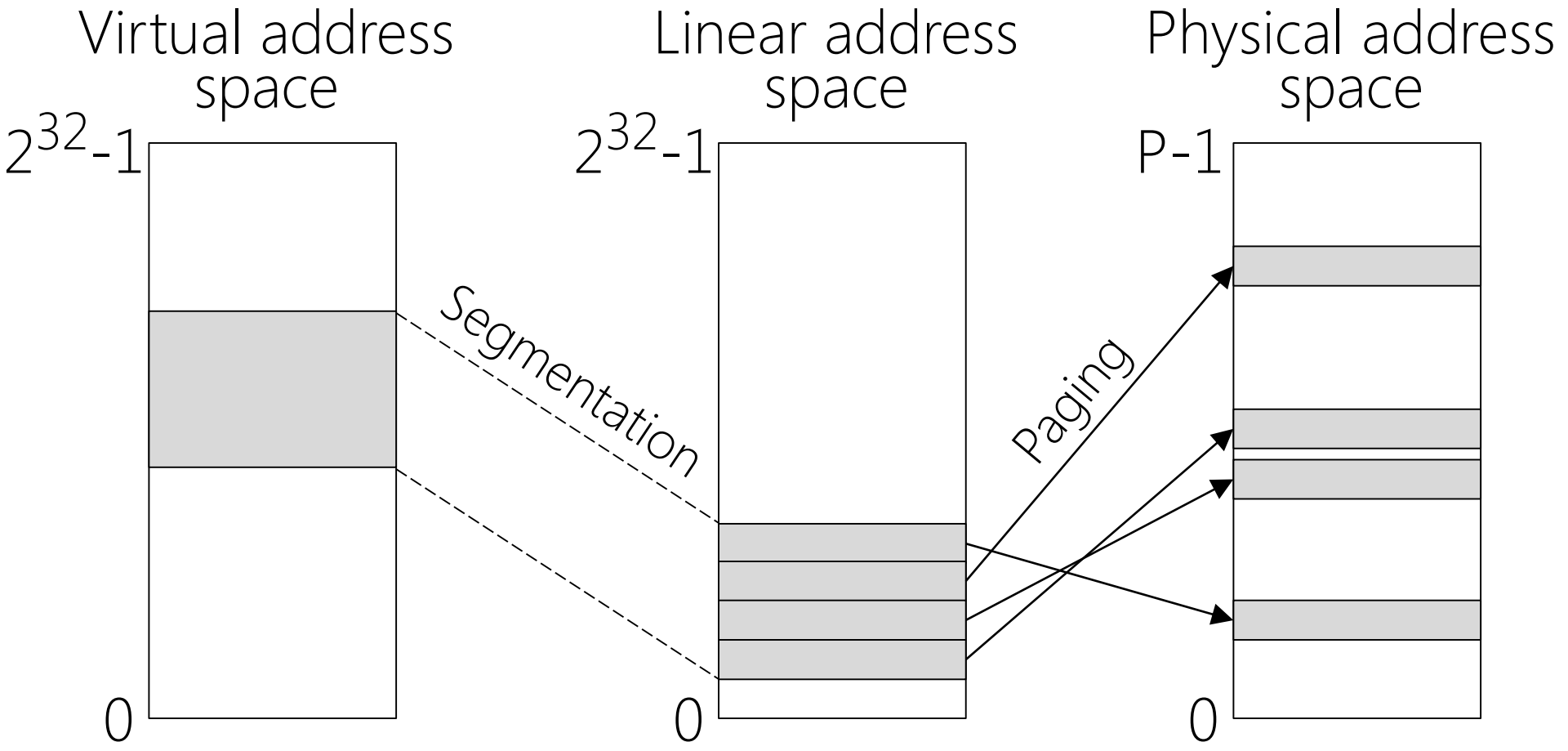


Note that:

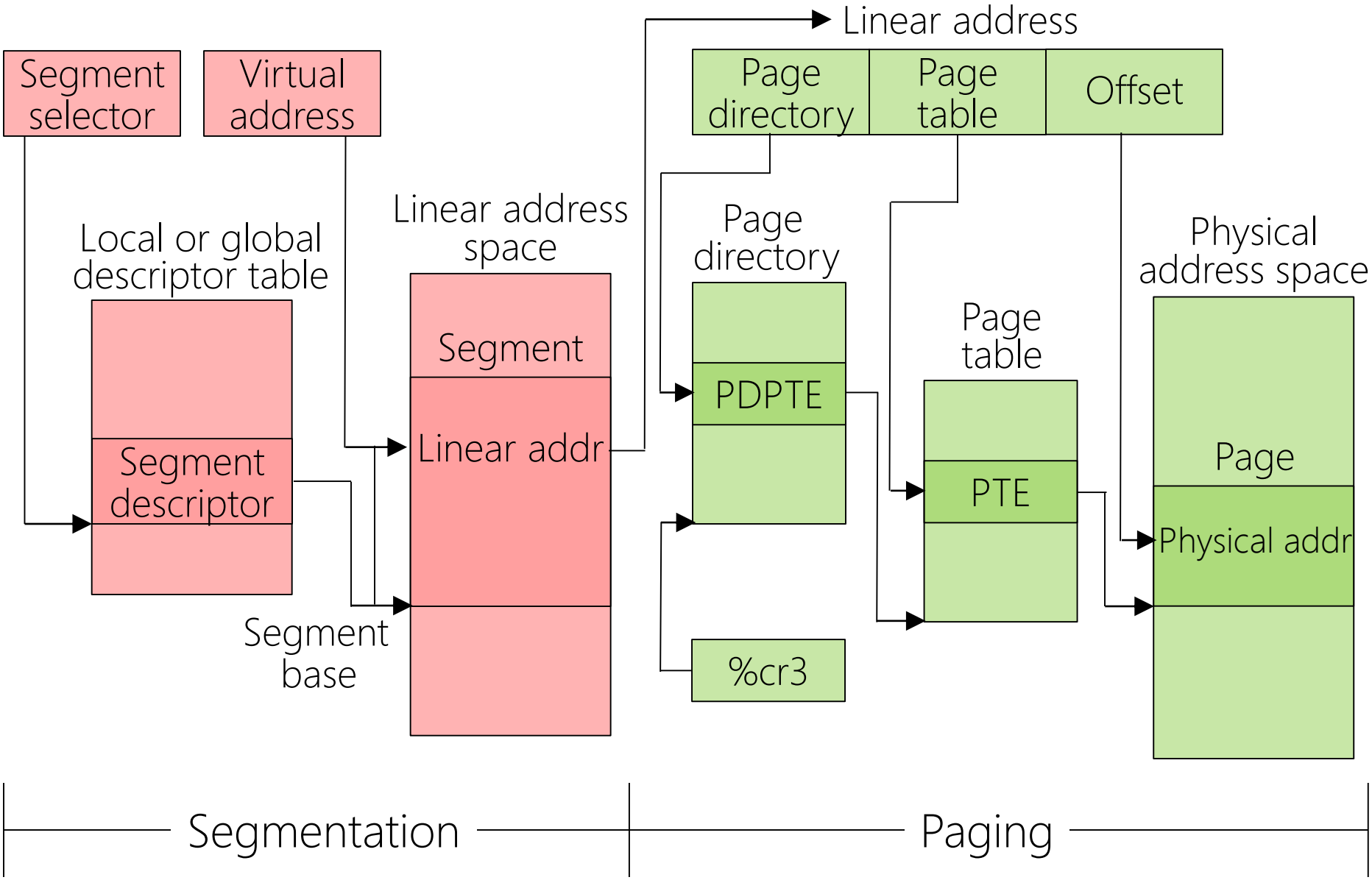
- Virtual address space is still 32-bits wide
- Physical address space is now 36-bits wide (i.e., only 36 bits of the 64-bit paging entries are actually used for physical addressing)

x86: Segmentation plus Paging

- x86 (32-bits) and x64 (when running in 32-bit mode) support both segmentation and paging!
- Strictly speaking, the "linear" address space is what gets paged

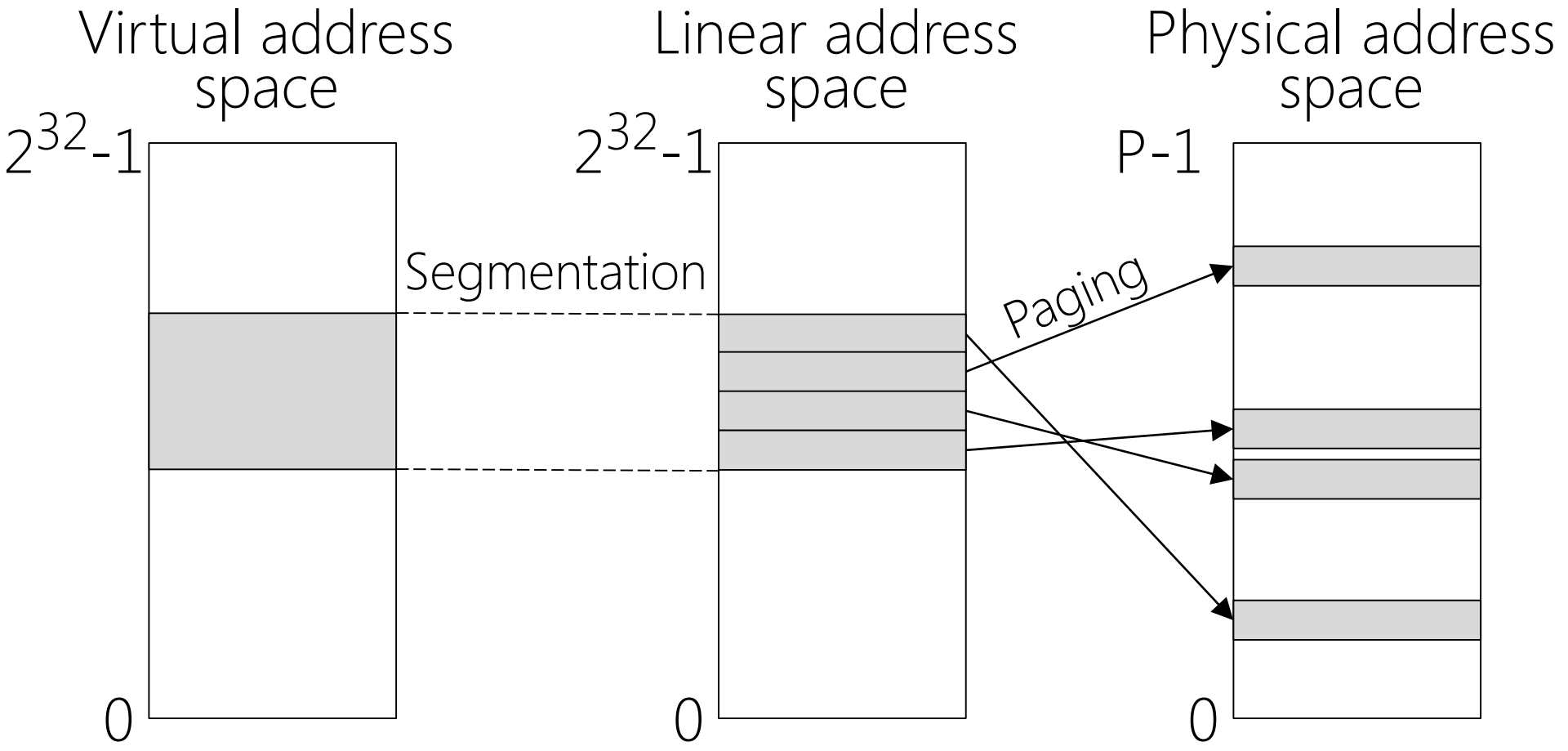


x86: Segmentation plus Paging



x86: Segmentation plus Paging

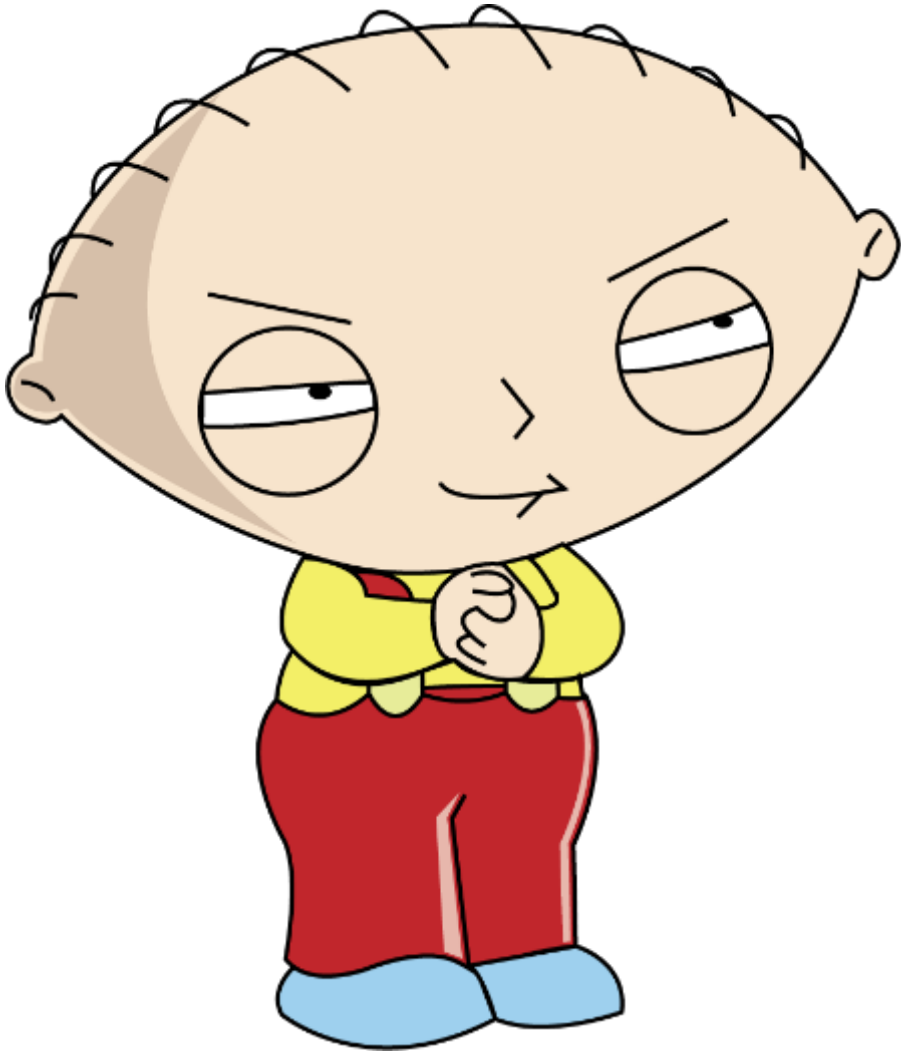
- Modern OSes like Windows and Linux configure **`%cs`**, **`%ds`**, and **`%ss`** to have a base of 0 and a bounds of 2^32 bytes
- So, segmentation is a no-op



x86: Segmentation plus Paging

- On 32-bit x86, modern OSes like Windows and Linux configure **`%cs`**, **`%ds`**, and **`%ss`** to have a base of 0 and a bounds of 2^32 bytes
- However, **`%fs`** and **`%gs`** used for systems chicanery
  - Ex: x86 Linux uses the **`%fs`** segment to store per-CPU information (remember that segment registers are per-core!); so, an instruction like **`inc %gs:(%eax)`** will increment a per-CPU memory location
- When x64 runs in 64-bit mode, the hardware forces **`%cs`**, **`%ds`**, and **`%ss`** to have a base of 0 and a bounds of 2^64 bytes
  - **`%fs`** and **`%gs`** still available for systems chicanery

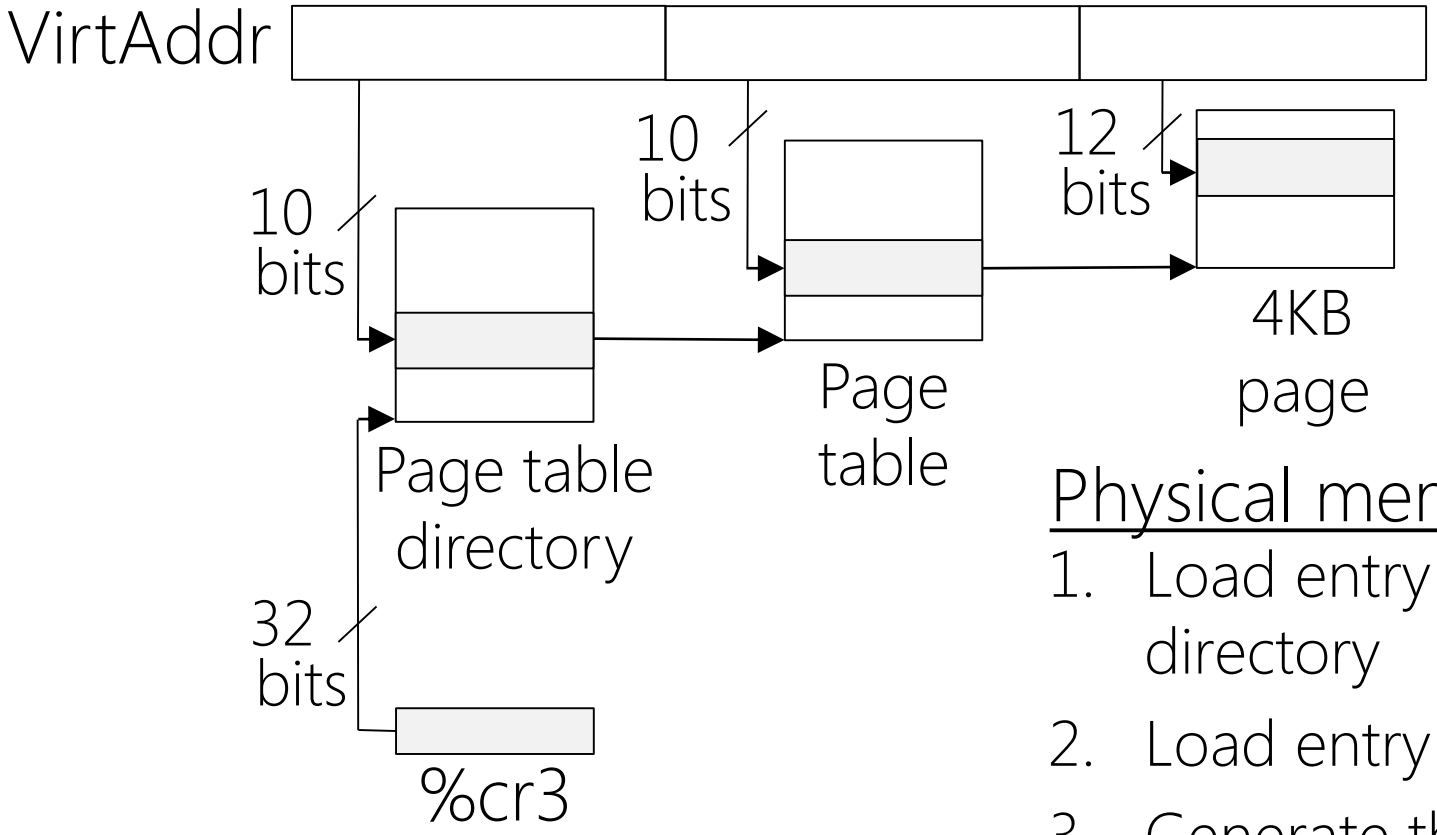
Q: What Do Page Tables Look Like On MIPS R3000?



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

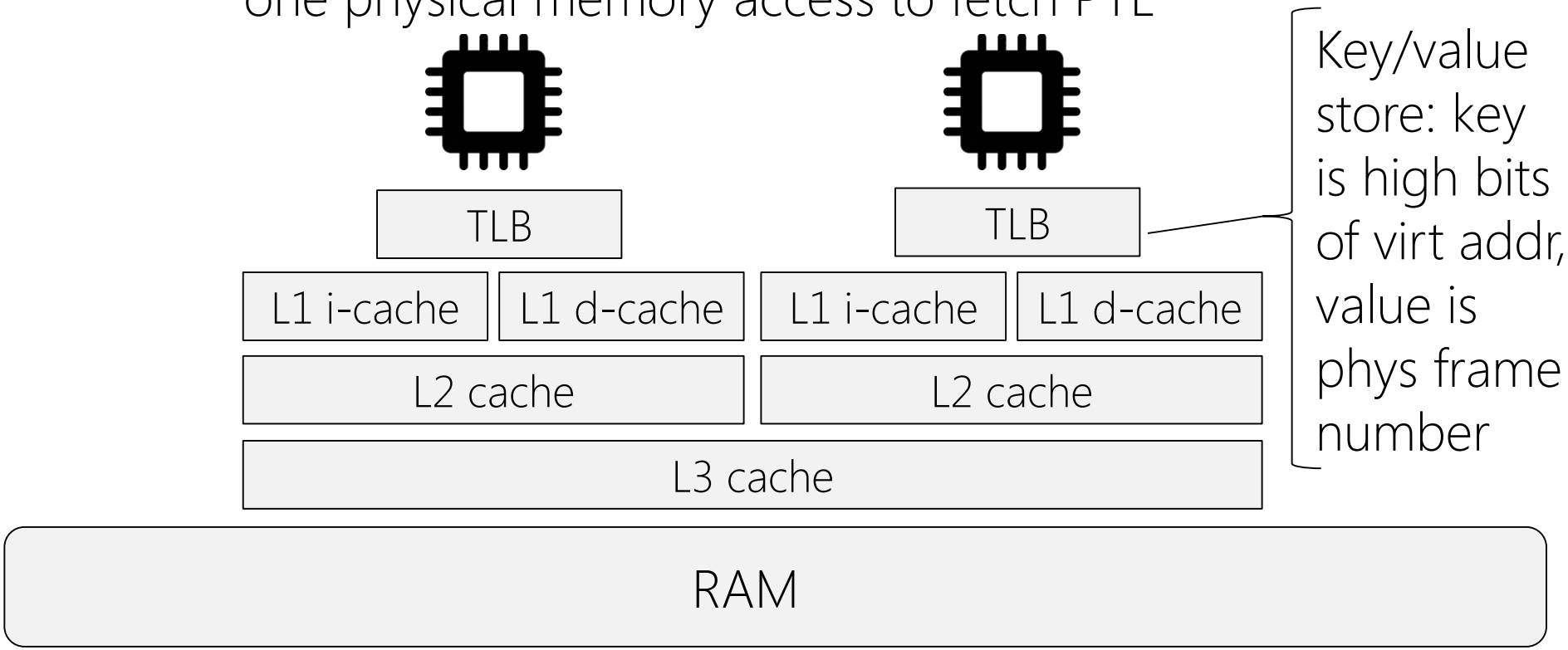


Physical memory accesses

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

Translation Lookaside Buffers (TLBs)

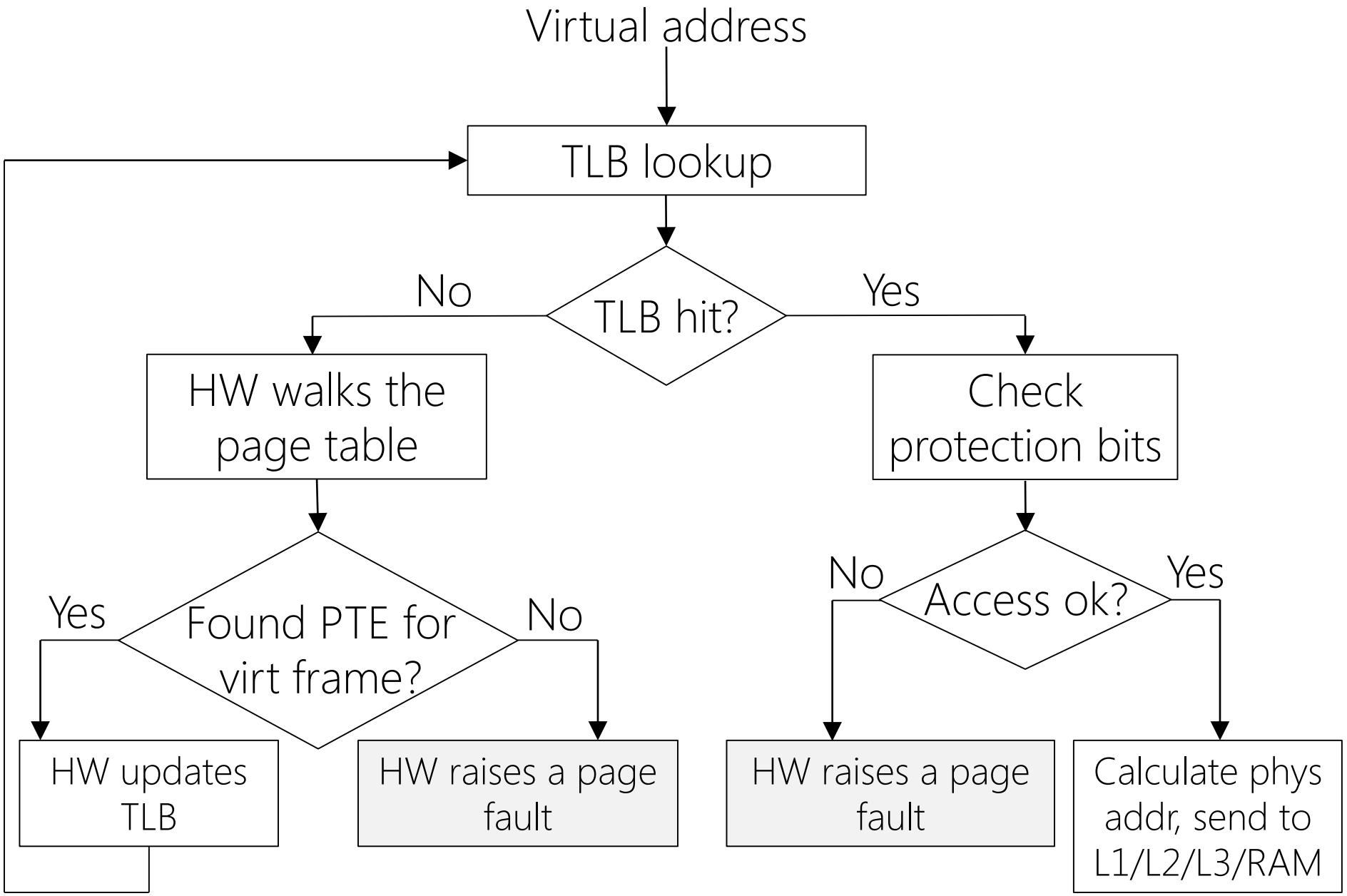
- Idea: Cache some PTEs in small hardware buffer
  - If virtual address has an entry in TLB, don't need to go to physical memory to fetch PTEs!
  - If virtual address misses in TLB, we must pay at least one physical memory access to fetch PTE



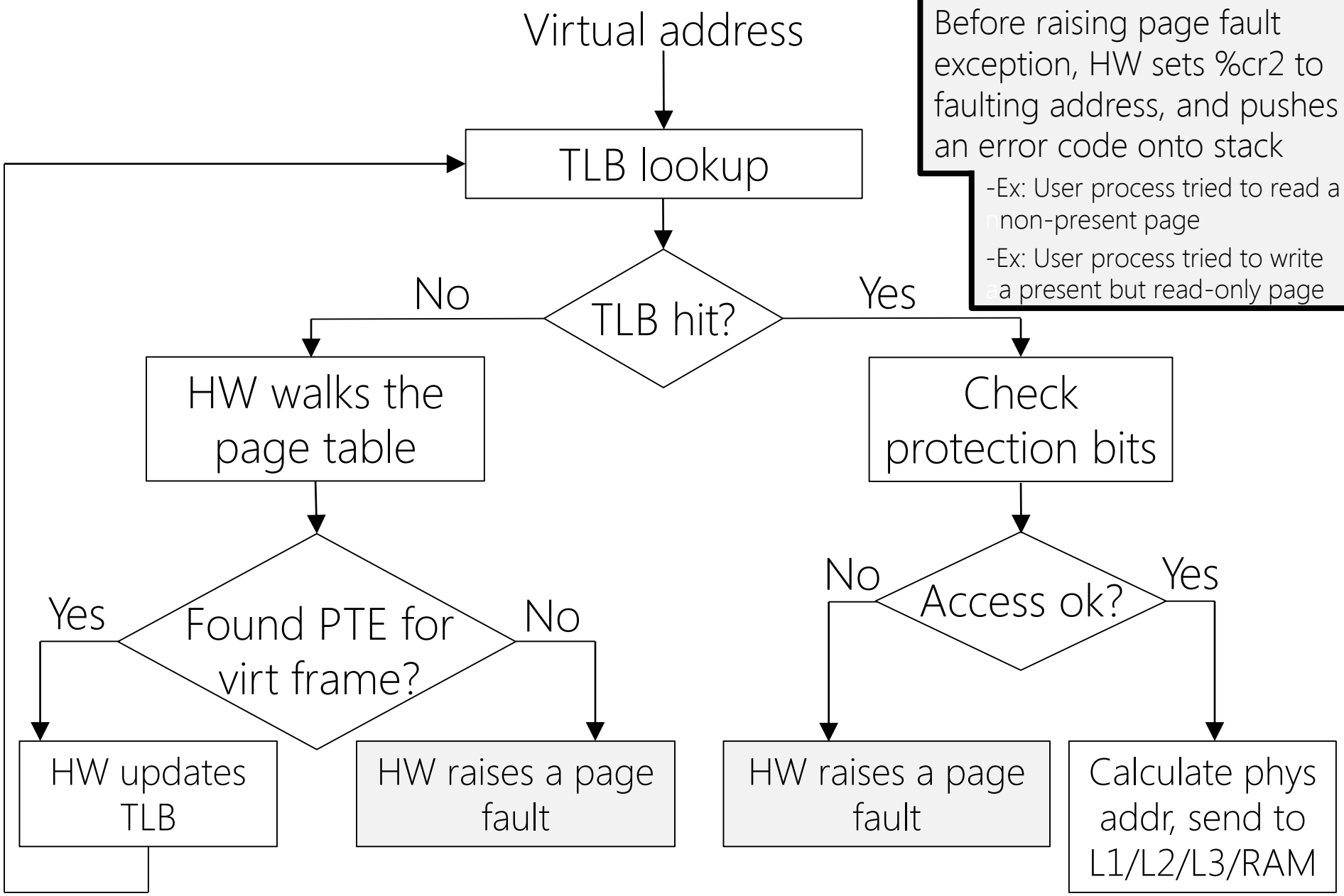
Translation Lookaside Buffers (TLBs)

- 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)

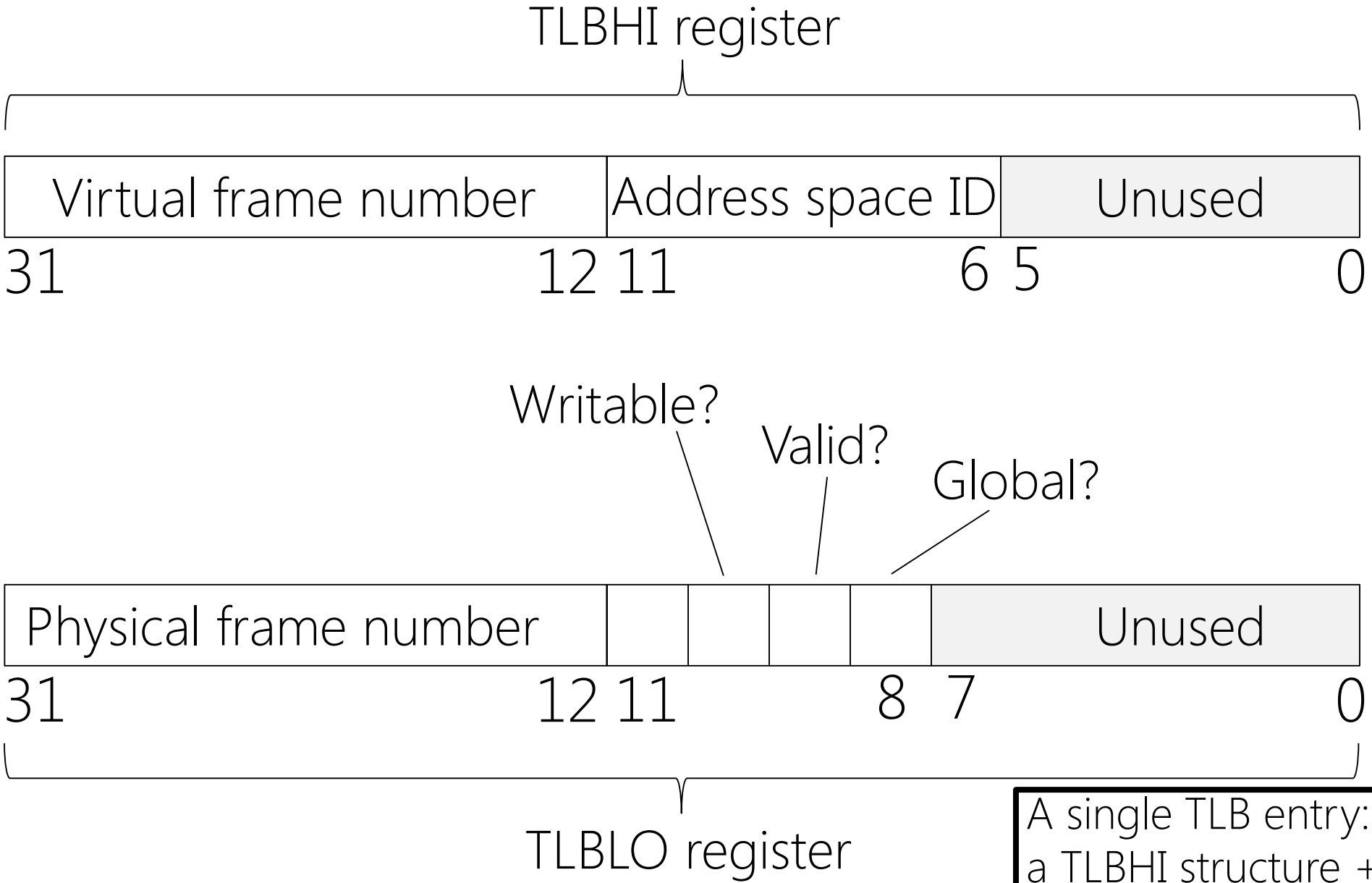
The Lifecycle of a Memory Reference on x86



The Lifecycle of a Memory Reference on x86

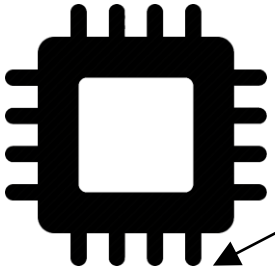


MIPS R3000: Interacting with the TLB

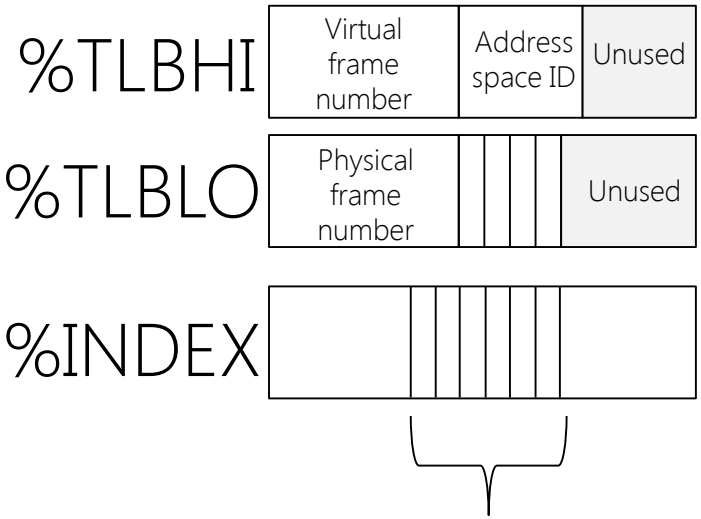


A single TLB entry:
a TLBHI structure +
a TLBLO structure

MIPS R3000: Interacting with the TLB

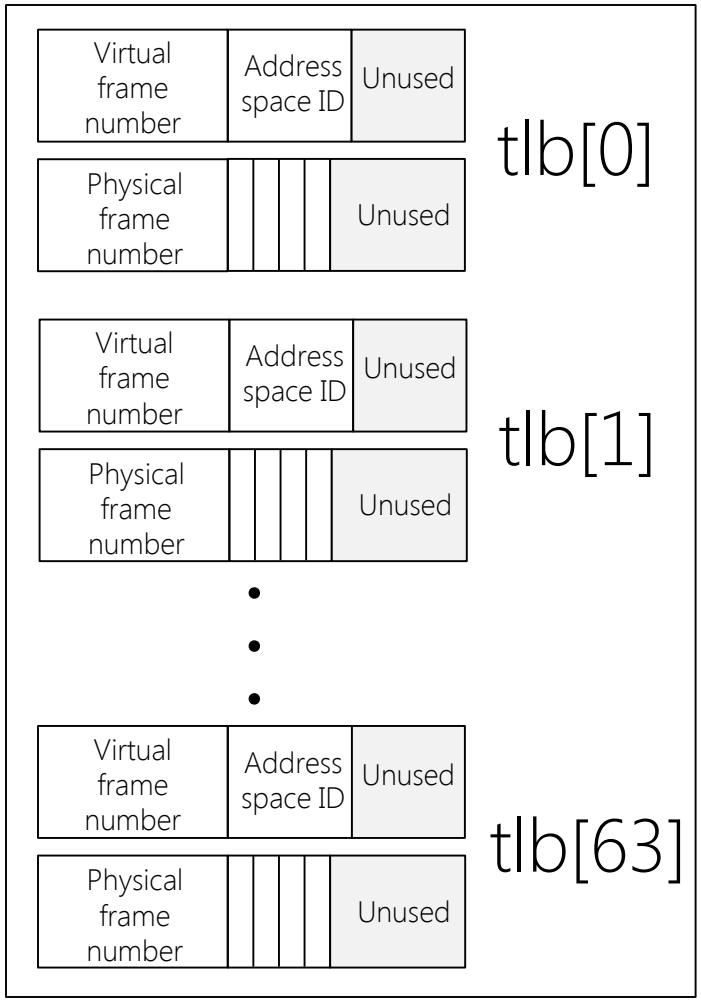


OS must set ASID during context switch!



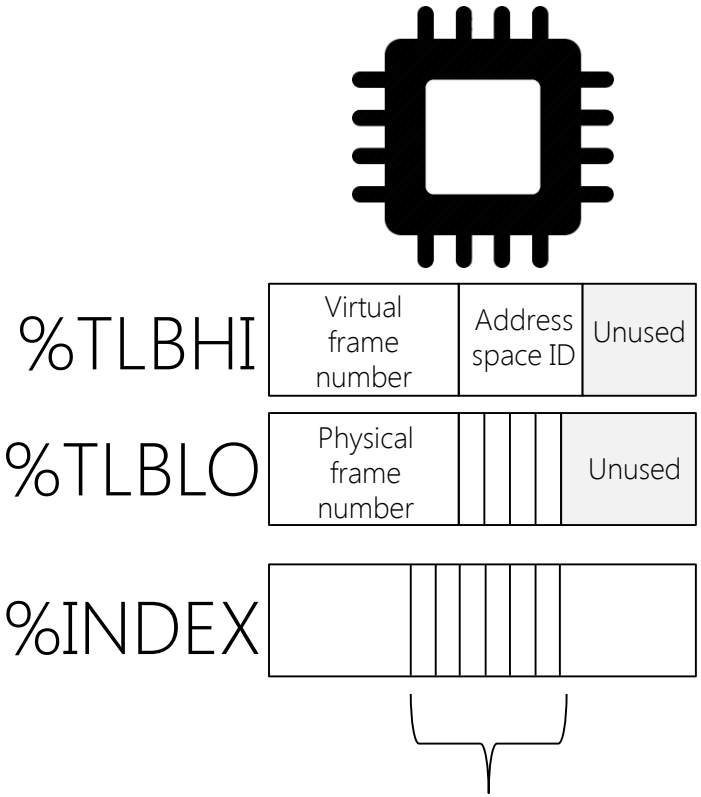
Used to select tlb[] entry for:

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



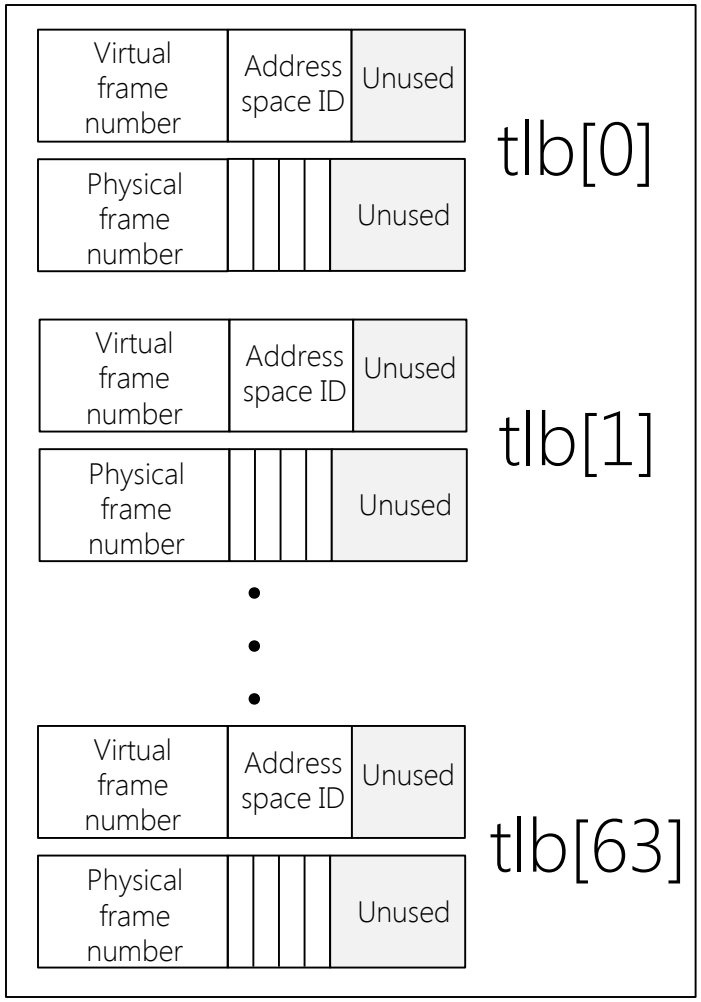
TLB

MIPS R3000: Interacting with the TLB



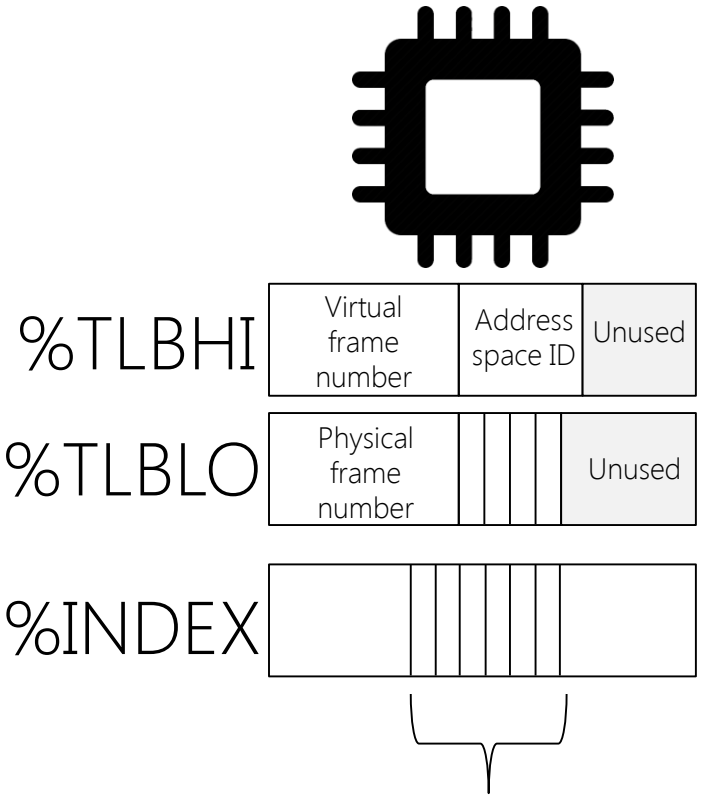
Set to tlb[] index by:

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



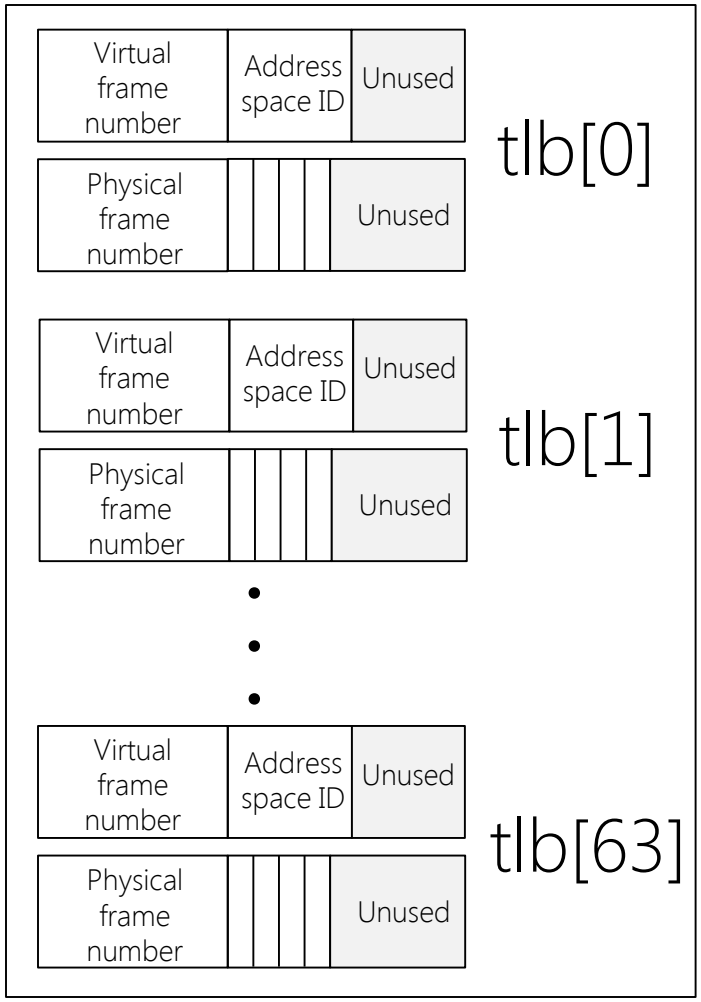
TLB

MIPS R3000: Interacting with the TLB

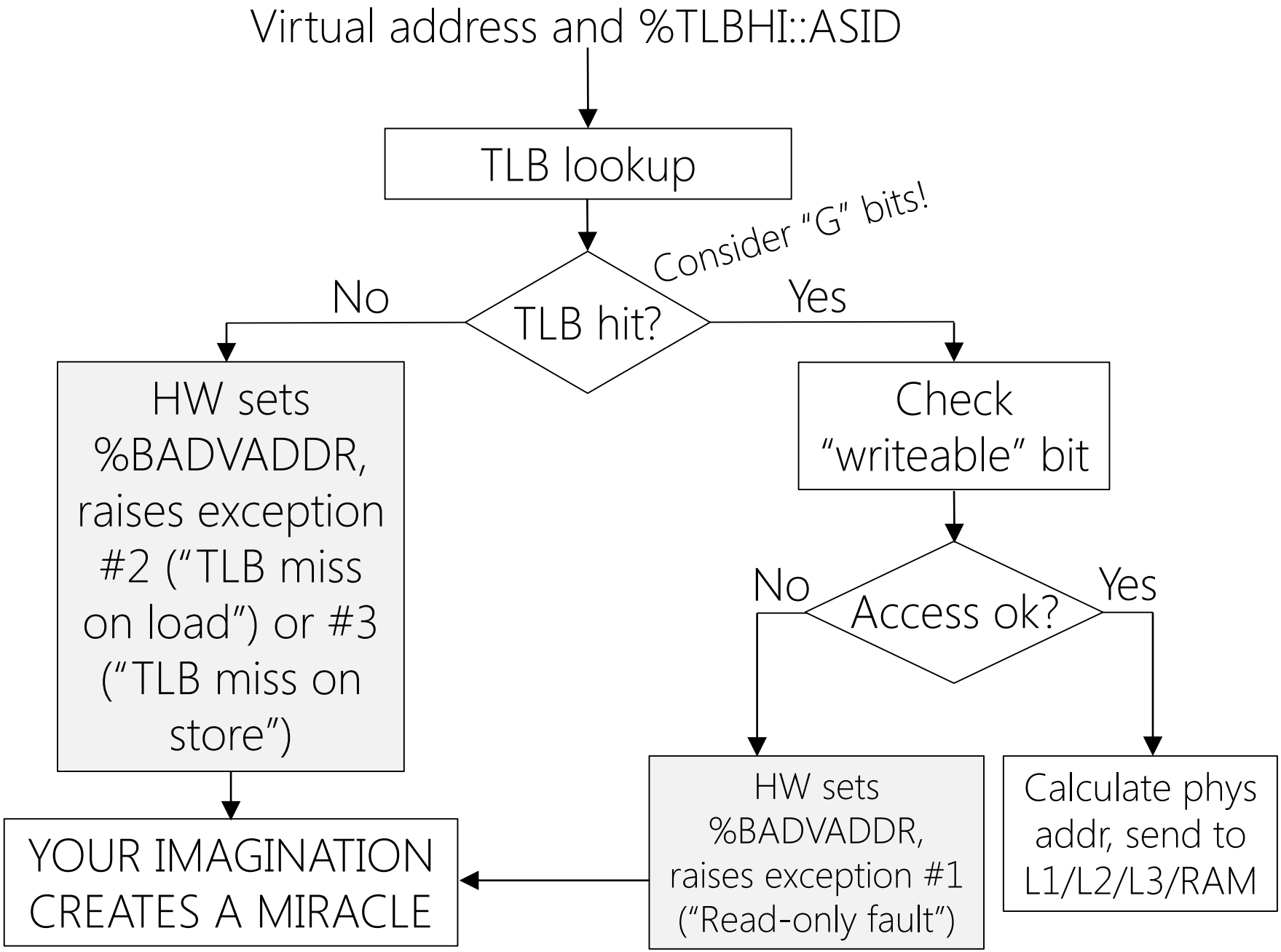


Not used by:

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



The Lifecycle of a Memory Reference on MIPS



TLBs and Context Switches

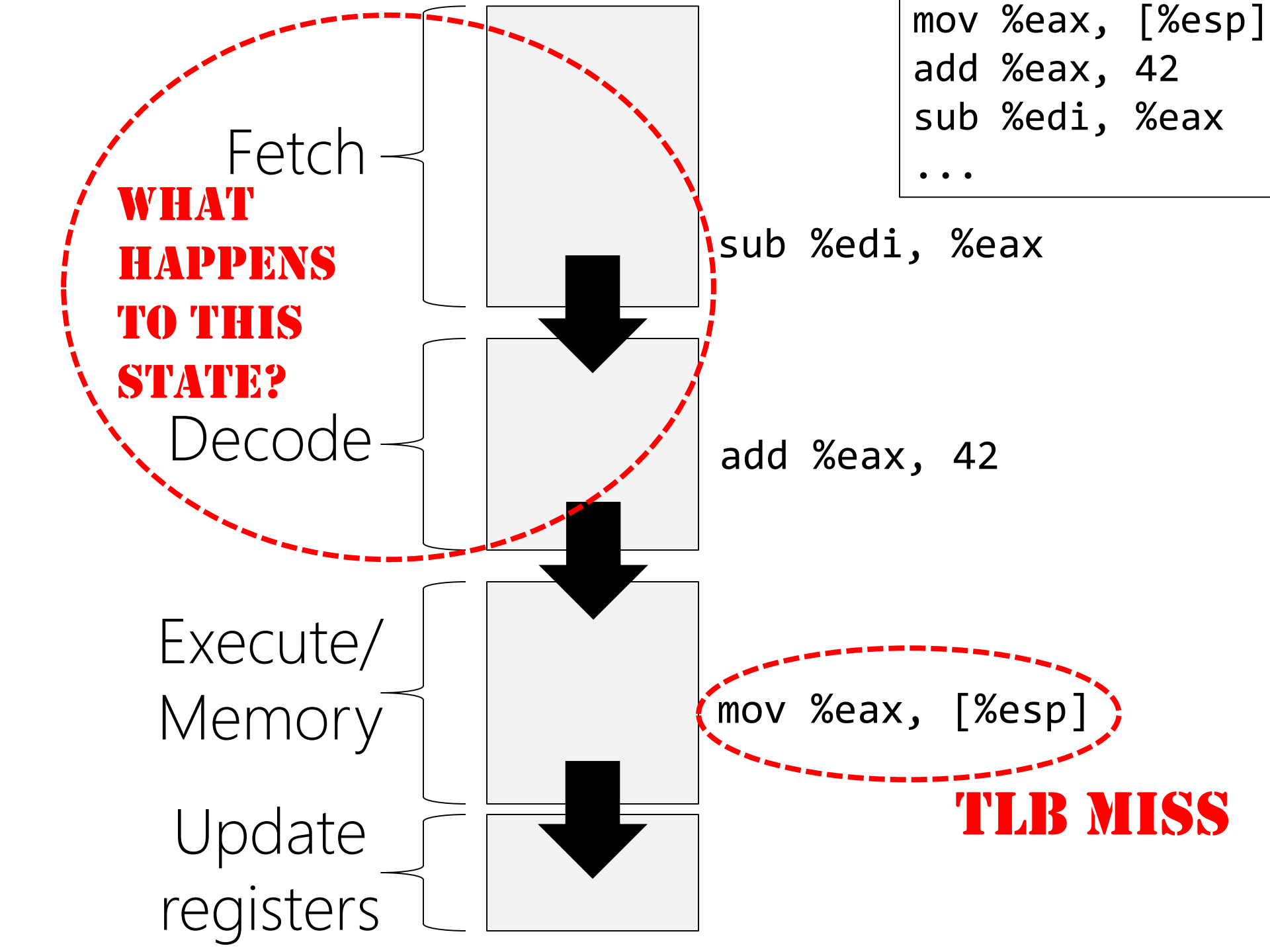
- If TLB entries are tagged with ASIDs:
  - OS updates current ASID (e.g., by setting TLBHI::ASID on MIPS)
  - OS doesn't need to flush TLBs
  - Even if OS occasionally has to evict entries, this is better than having to evict ALL entries during EVERY context switch (since this generally requires size(TLB) 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


mov %eax, [%esp]
add %eax, 42
sub %edi, %eax
...
Fetch
WHAT HAPPENS TO THIS STATE?
sub %edi, %eax
Decode
add %eax, 42
Execute/ Memory
mov %eax, [%esp]
TLB MISS
Update registers

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 format can't be changed by OS