Virtual Memory

CSCI 237: Computer Organization
31st Lecture, Wednesday, November 20, 2024

Kelly Shaw

Slides originally designed by Bryant and O'Hallaron @ CMU for use with Computer Systems: A Programmer's Perspective, Third Edition

Administrative Details

- Read CSAPP 9.3-9.6 (Ch. 9 sections are short)
- Lab #5 due today at 11pm
- Glow quiz opens today at 2:30pm and due Friday at 2:30pm
- Lab #6 partner signup due Friday at 8am
- Colloquium talk on Friday at 2:35pm in Wege
  - Jeremy Fineman, Georgetown
  - ” Beating Bellman-Ford: Faster Single-Source Shortest Paths with Negative Weights”

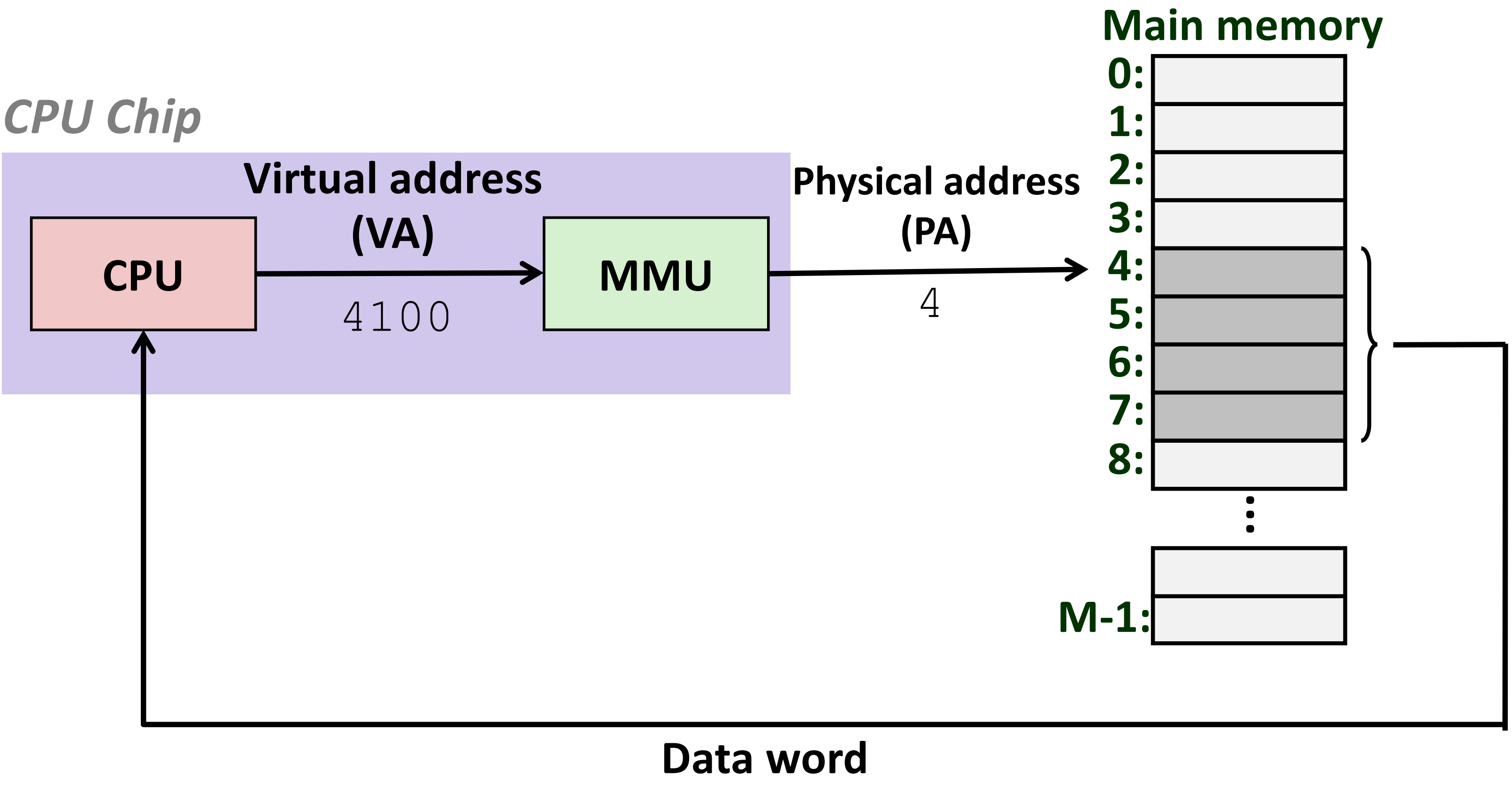
Last Time: Virtual Memory

- Caches in real systems
- Performance impact of caches
  - The memory mountain
- Address spaces (Ch 9.2)
- VM as a tool for caching (Ch 9.3)
- VM as a tool for memory management (Ch 9.4)
- VM as a tool for memory protection (Ch 9.5)

Today: Virtual Memory

- Address translation (Ch 9.6)
- End-to-end example of a simple memory system

Review: A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

Today: Virtual Memory

- Address translation (Ch 9.6)
- End-to-end example of a simple memory system

VM Address Translation (formally)

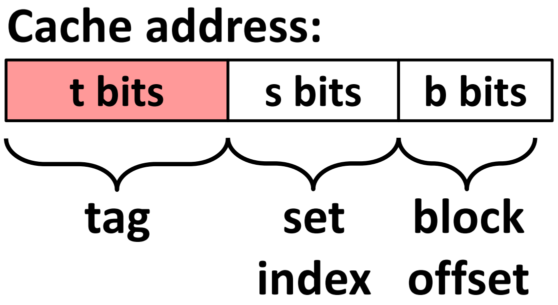
- Virtual Address Space
  - $V = \{0, 1, \ldots, N\text{–}1\}$
- Physical Address Space
  - $P = \{0, 1, \ldots, M\text{–}1\}$
- Address Translation. ***MAP:* $V \rightarrow P \cup \{\varnothing\}$**
  - For virtual address ***a***:
    - ***MAP(a) = a′*** if data at virtual address ***a*** in ***V*** is at physical address ***a′*** in ***P***
    - ***MAP(a) = $\varnothing$*** if data at virtual address ***a*** is not in physical memory (either unallocated or stored on disk)

Summary of Address Translation Jargon

- Basic Parameters
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (in bytes) of physical and virtual pages
- Components of the virtual address (VA)
  - **VPN**: Virtual page number
  - **VPO**: Virtual page offset
  - **TLBI**: TLB (Translation Lookaside Buffer) index
  - **TLBT**: TLB tag
- Components of the physical address (PA)
  - **PPN:** Physical page number
  - **PPO**: Physical page offset (same as VPO)

Translating Virtual to Physical Addresses

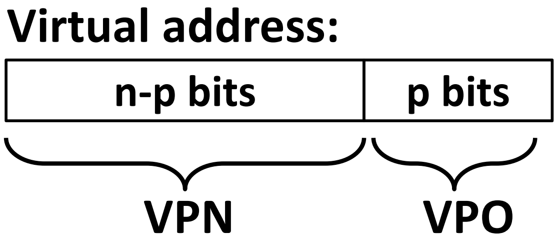
- Control register (CR3) stores physical address of Page Table
- Given the page table's location, how does lookup work?
  - Familiar approach: break up the address and index into our cache



Cache address:

t bits | s bits | b bits

tag | set index | block offset



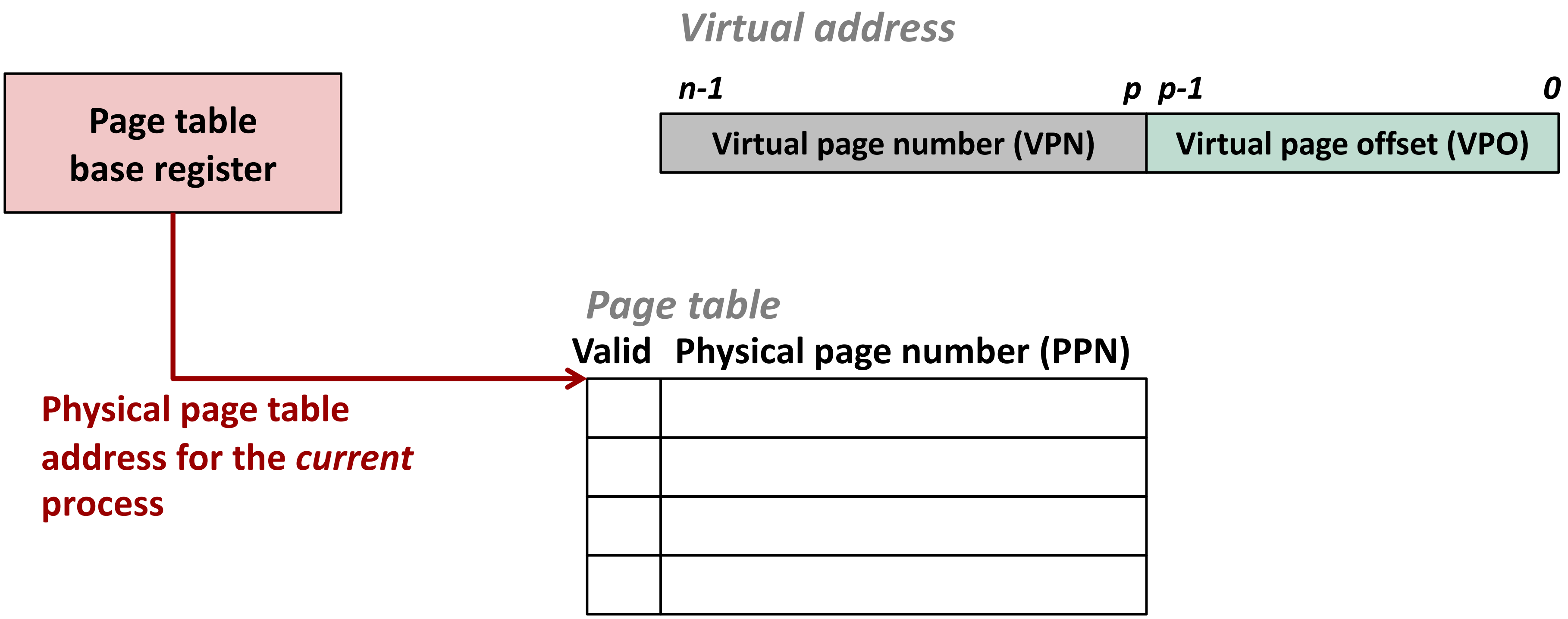
Virtual address:

n-p bits | p bits

VPN | VPO

Typically, how many bits is p?	**$\log_2(4096)$ => 12 bits**
What does the VPN tell us?	**Offset of our PTE in the page table**
What does the VPO tell us?	**Offset of our data within the page**

Address Translation With a Page Table



Virtual address

n-1 *p* *p-1* *0*

Virtual page number (VPN)

Virtual page offset (VPO)

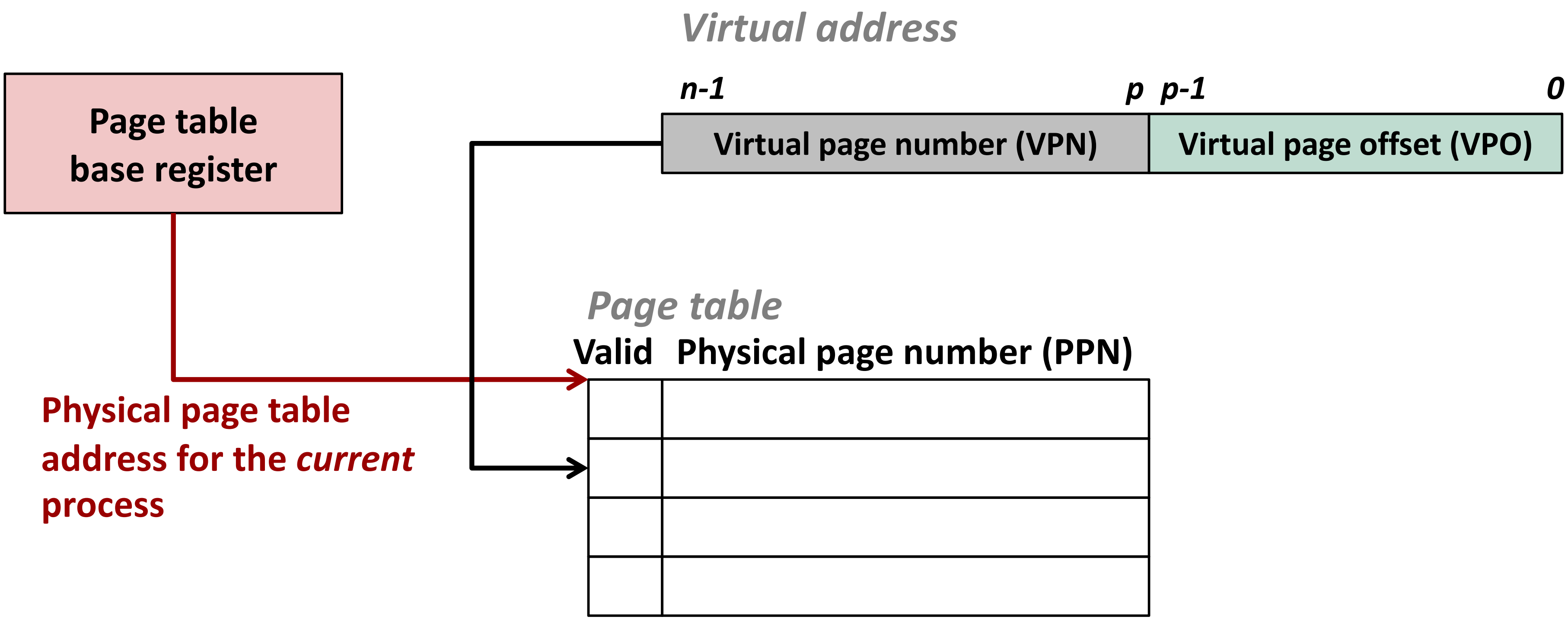
Page table base register

Page table

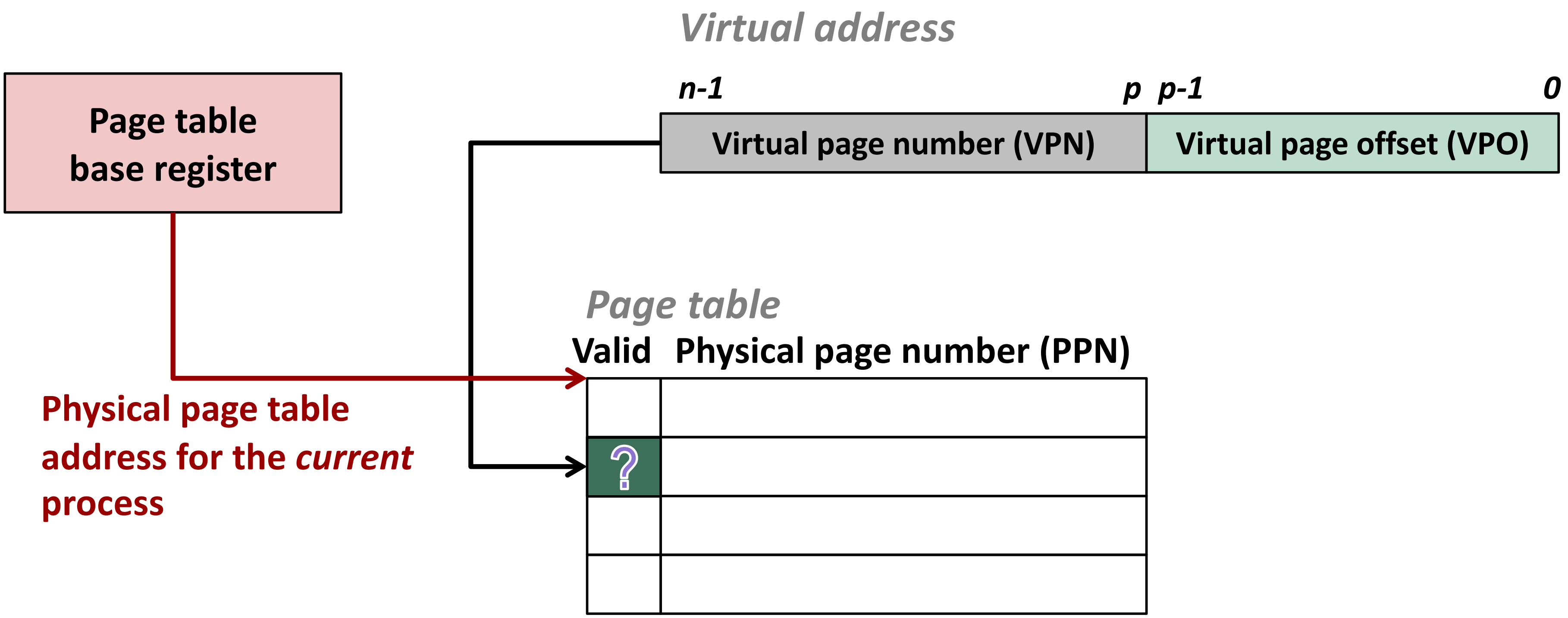
Valid Physical page number (PPN)

Physical page table address for the *current* process

Address Translation With a Page Table

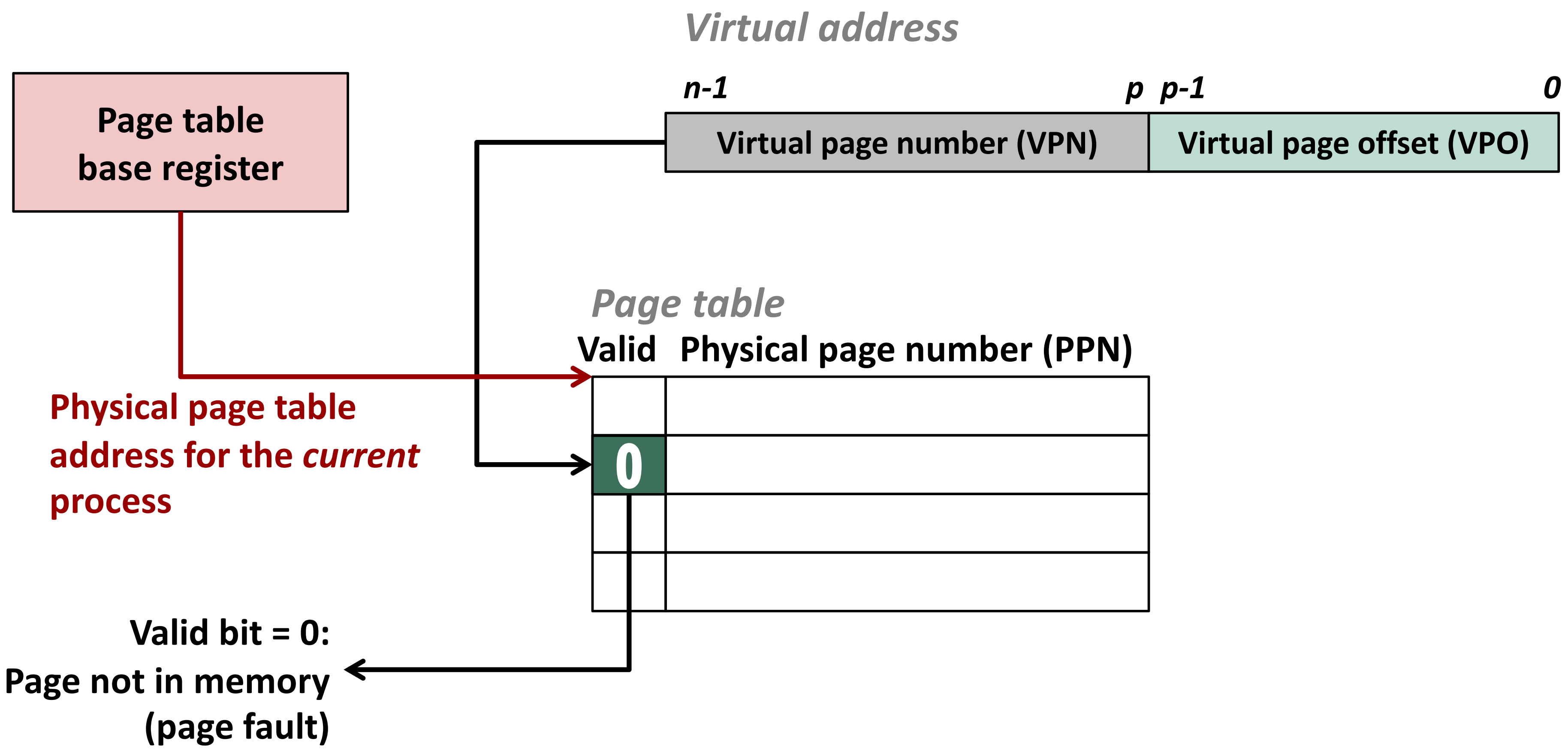

Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table
base register
Page table
Valid
Physical page number (PPN)
Physical page table
address for the *current*
process

Address Translation With a Page Table


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table
base register
Page table
Valid
Physical page number (PPN)
?

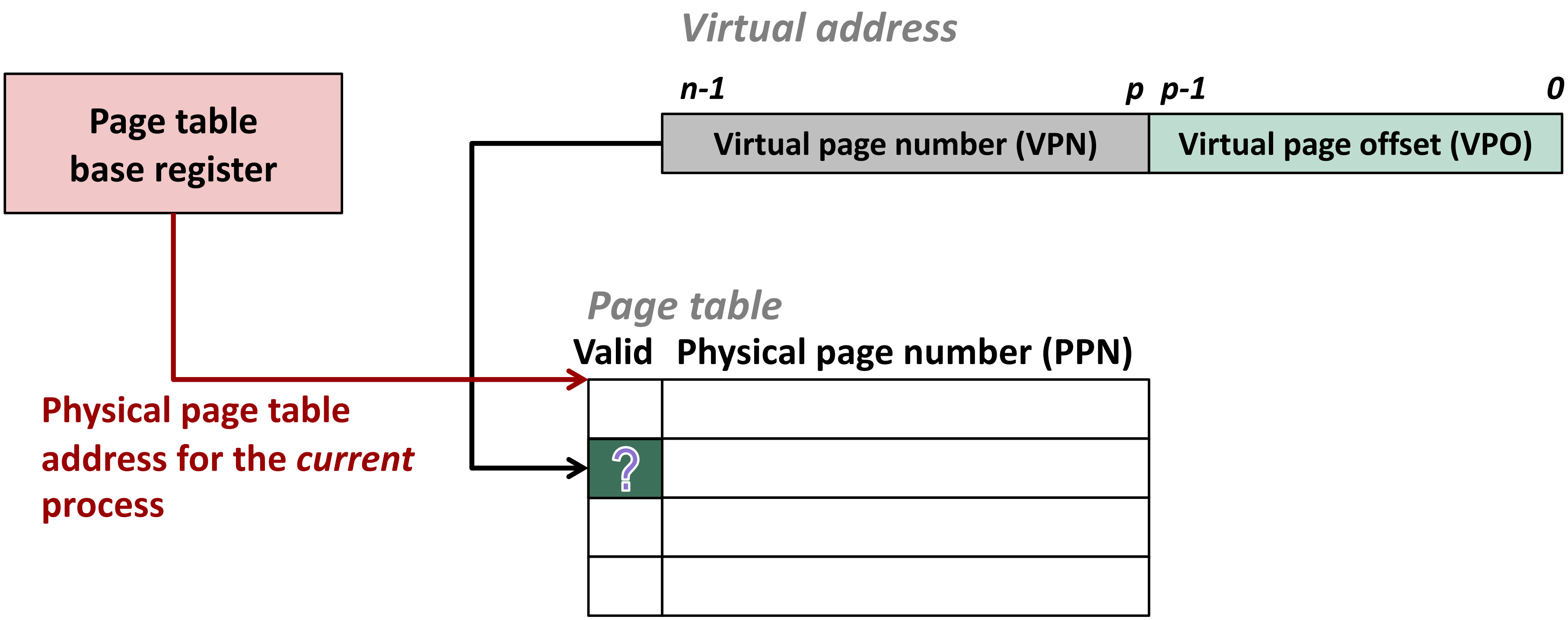
Physical page table address for the *current* process

Address Translation With a Page Table


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register
Page table
Valid
Physical page number (PPN)
0
Physical page table address for the *current* process
Valid bit = 0:
Page not in memory
(page fault)

What do we do on a page fault? **Page fault exception handler reads physical page from disk and updates PTE for this VA.**

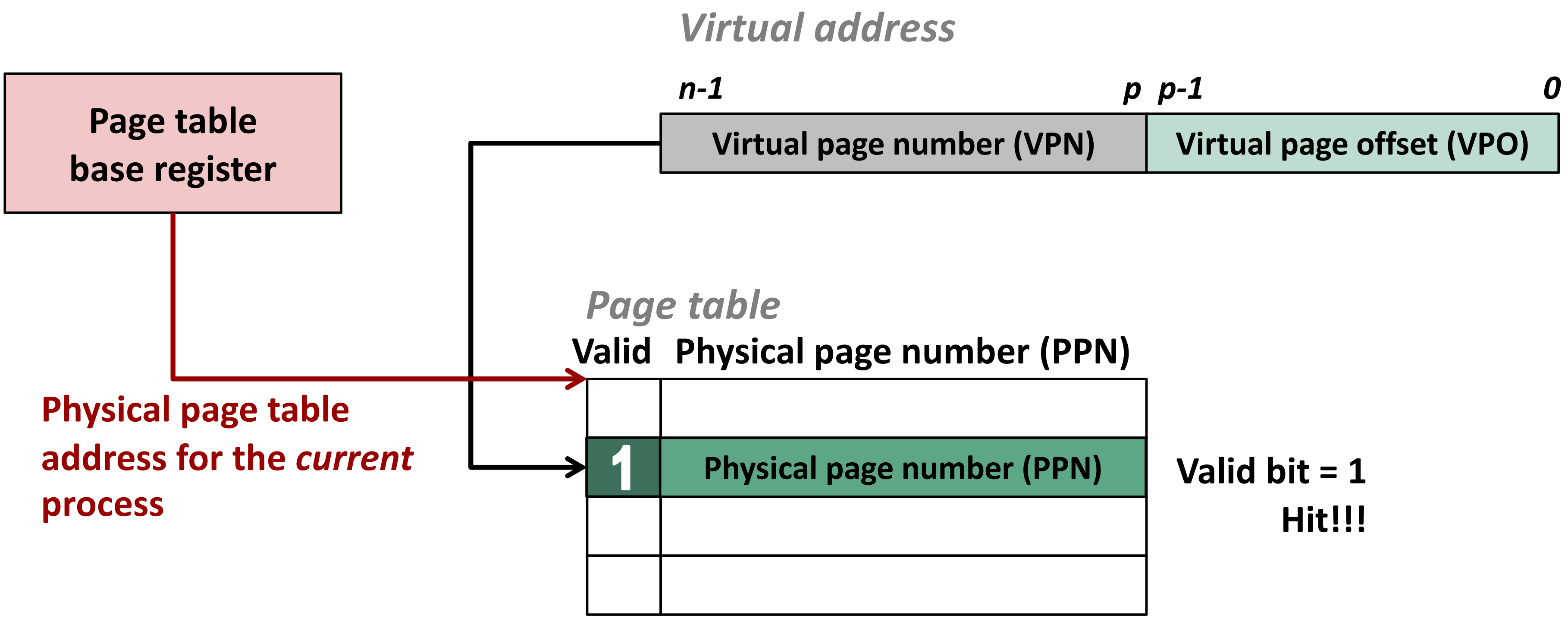
Address Translation With a Page Table



Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register
Page table
Valid
Physical page number (PPN)
?

Physical page table address for the *current* process

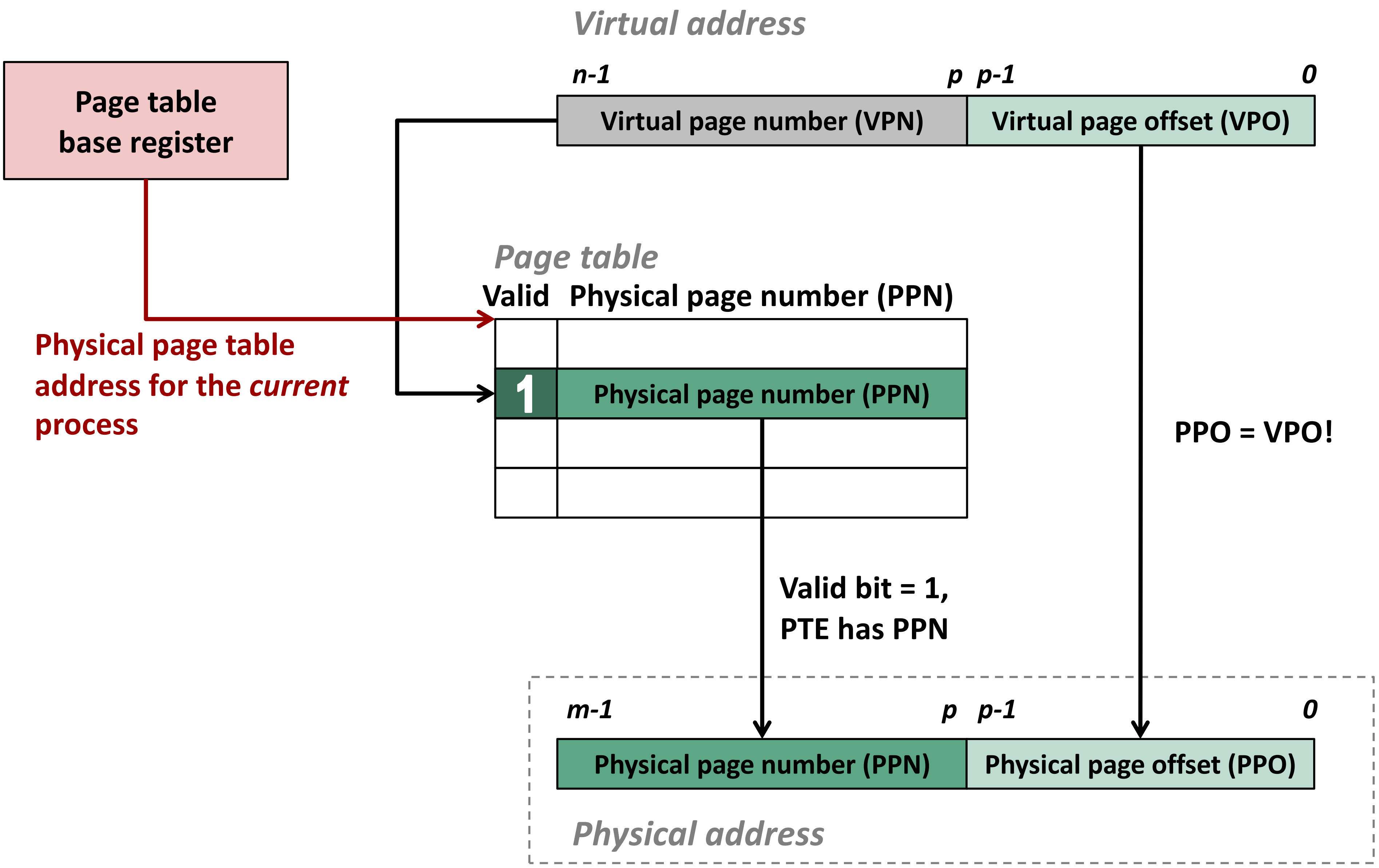
Address Translation With a Page Table


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table
base register
Physical page table
address for the *current*
process
Page table
Valid
Physical page number (PPN)
1
Physical page number (PPN)
Valid bit = 1
Hit!!!

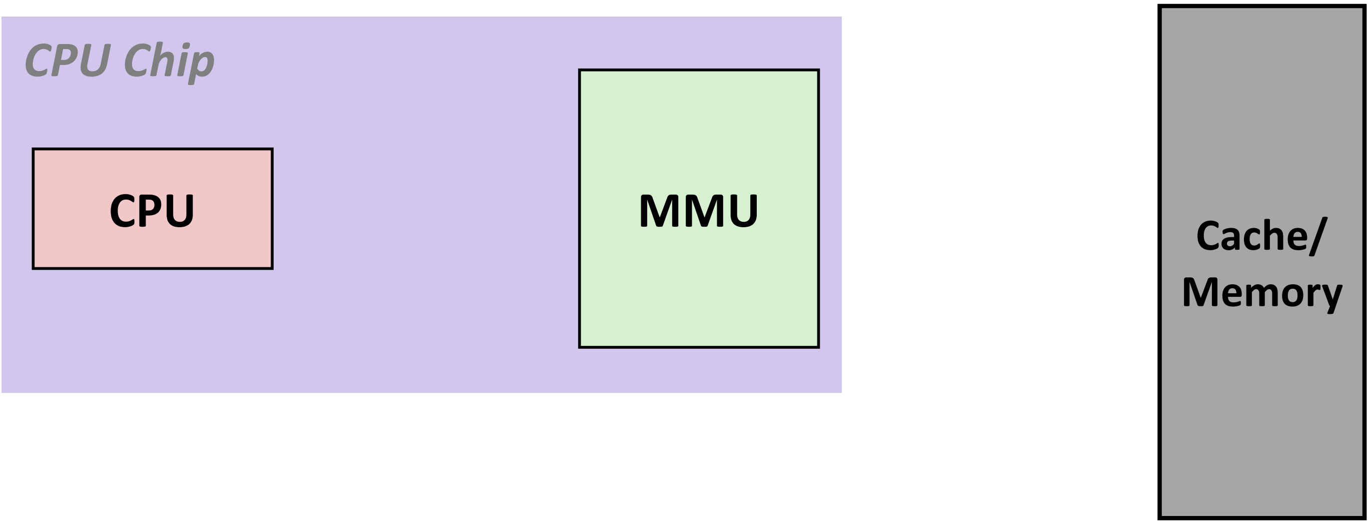
What does a hit mean? **The PTE contains the PPN.**

Are we done? **No. A physical address consists of a PPN and a PPO. We still need the PPO…**

Address Translation With a Page Table

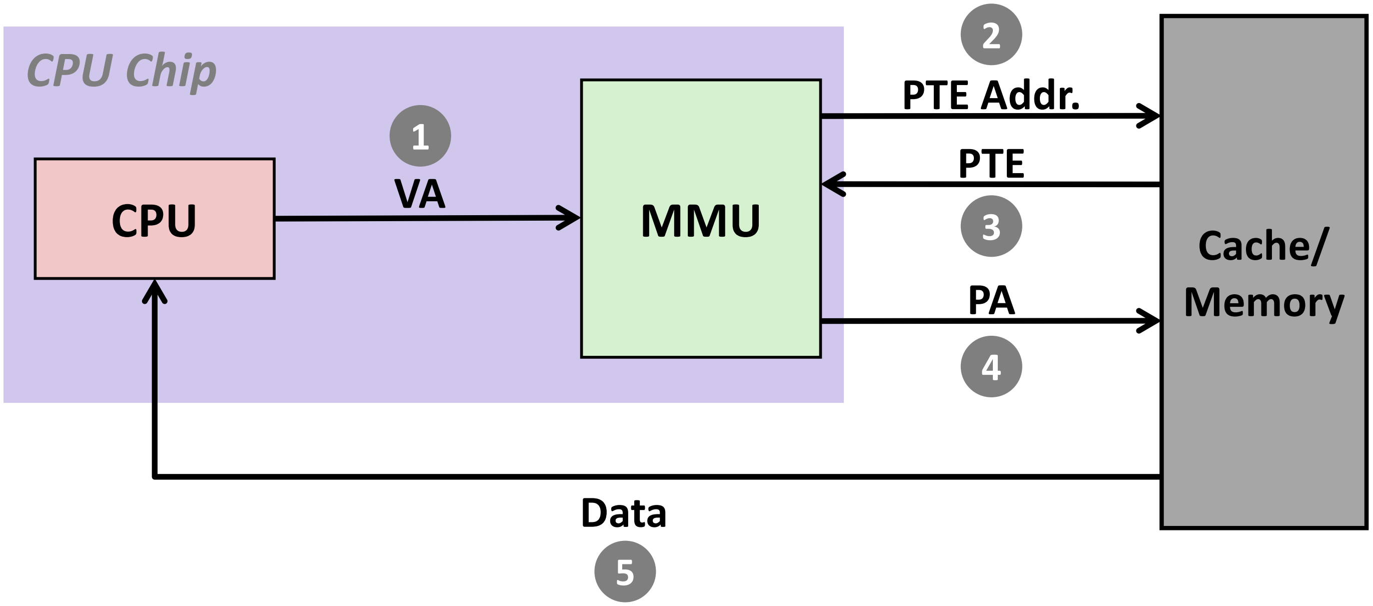

Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table
base register
Physical page table
address for the *current*
process
Page table
Valid
Physical page number (PPN)
1
Physical page number (PPN)
PPO = VPO!
Valid bit = 1,
PTE has PPN
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Putting it all together: Address Translation



- The CPU generates a request for a virtual address
- The MMU drives the page translation
- Let’s explore the interaction between the CPU, MMU, and Cache/memory on:
  - A page table hit
  - A page fault (miss)

Address Translation: Page Hit



1) Processor sends virtual address to MMU

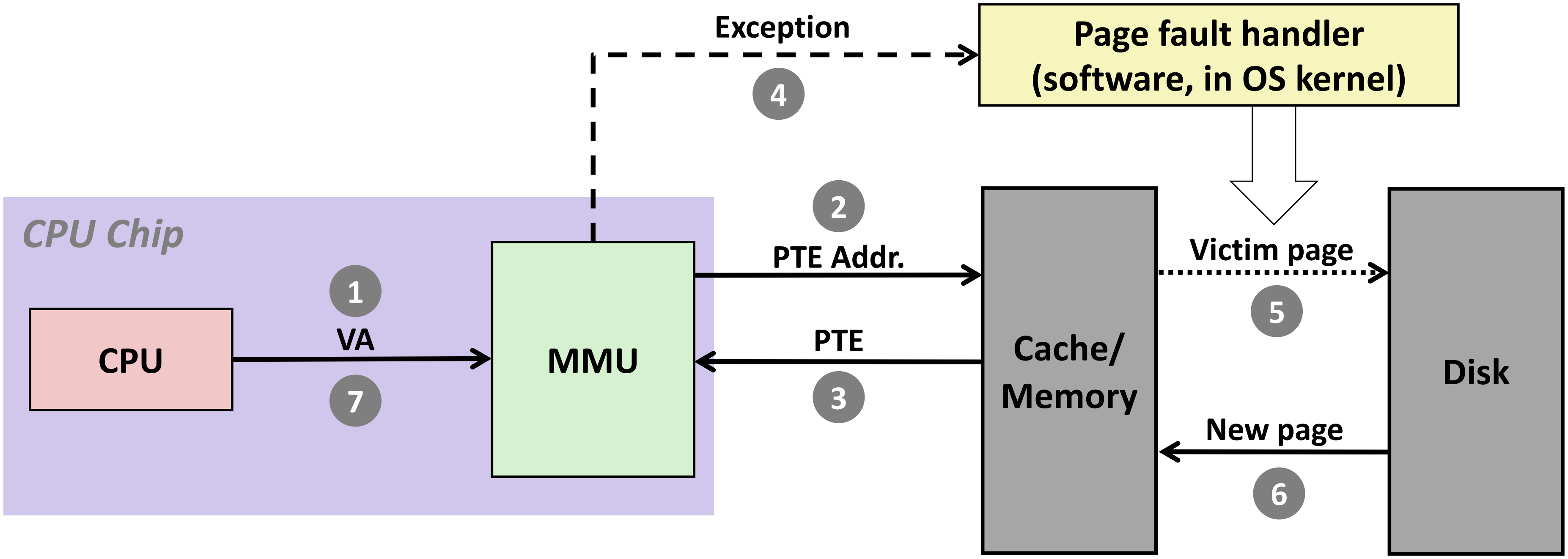
2-3) MMU fetches PTE from page table in cache/memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

Page hit handled entirely by hardware!

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in cache/memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim PTE (and, if dirty, swaps it out to disk)

6) Handler pages in new page and updates PTE in memory

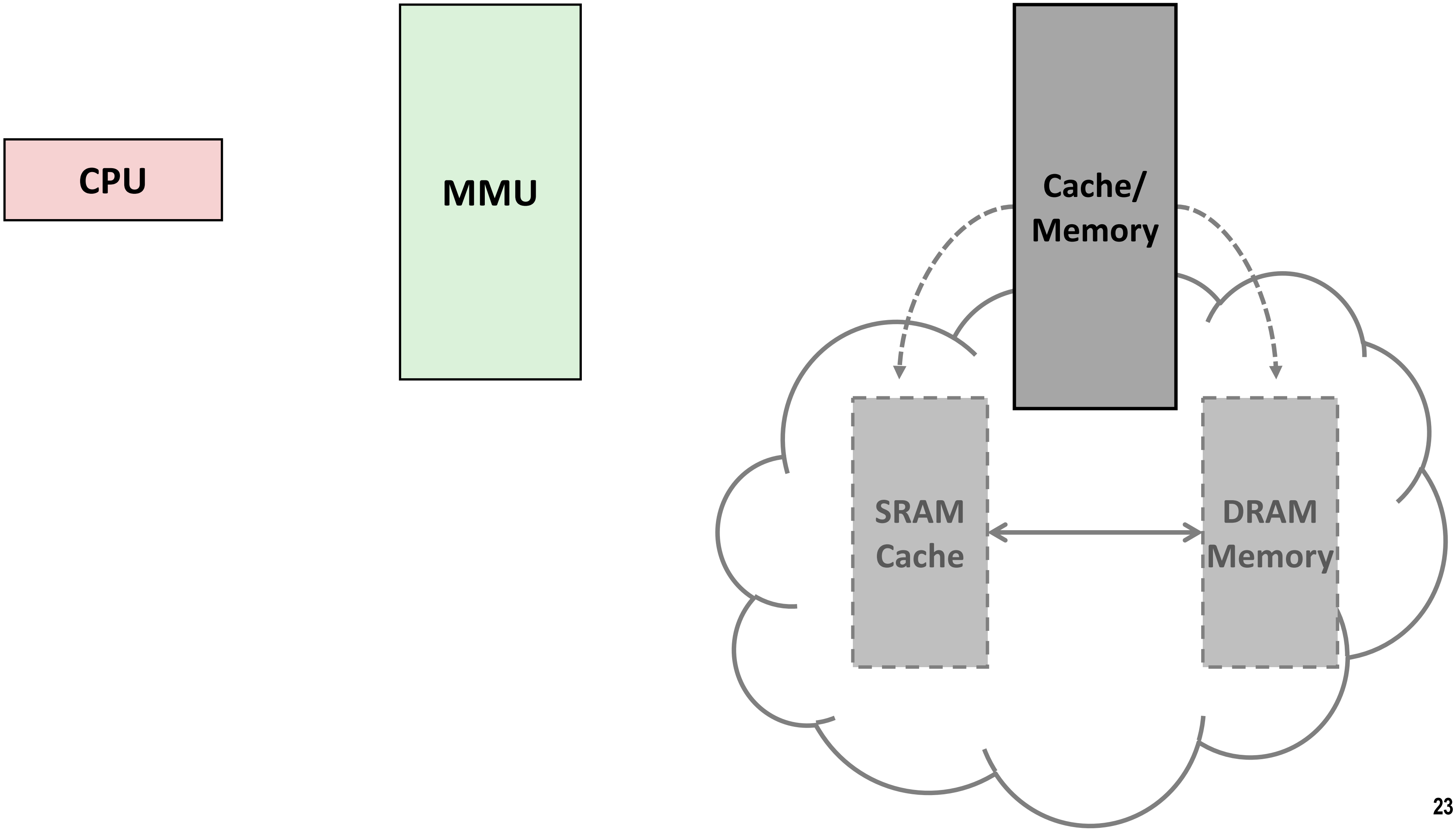
7) Handler returns to original process. Original process restarts faulting instruction

Restarted instruction should now hit.

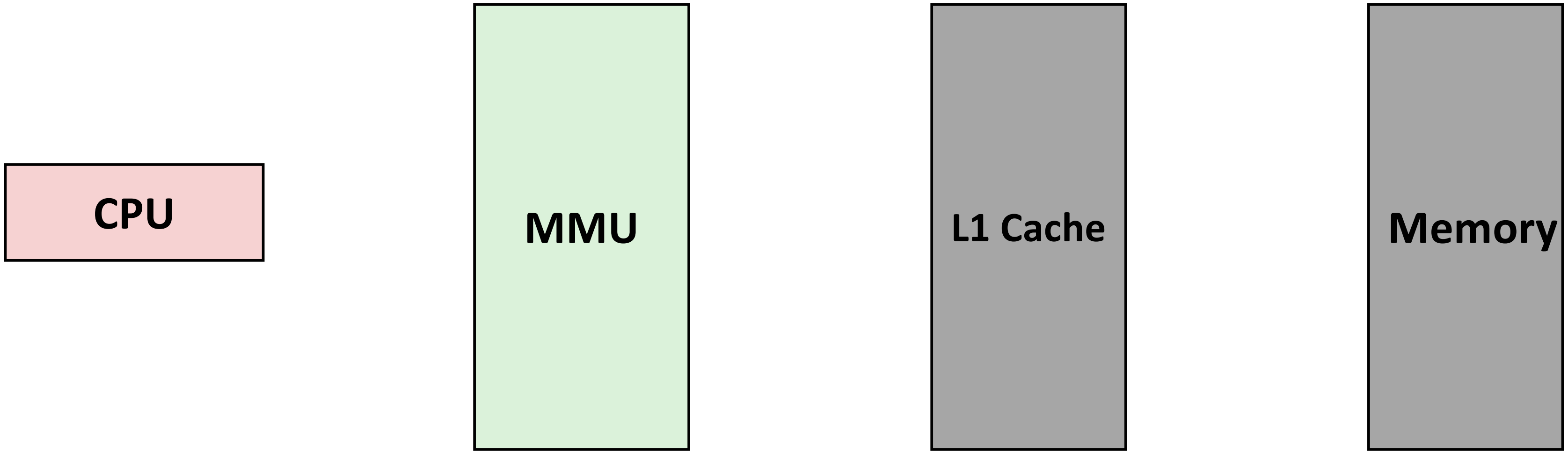
Practice on Your Own

- Suppose a system uses 2048B sized pages and addresses are specified using 32 bits. How many PTEs would be needed in the page table for a process? How many bits would be need to specify the virtual page offset (i.e., which byte in the page is being accessed)?

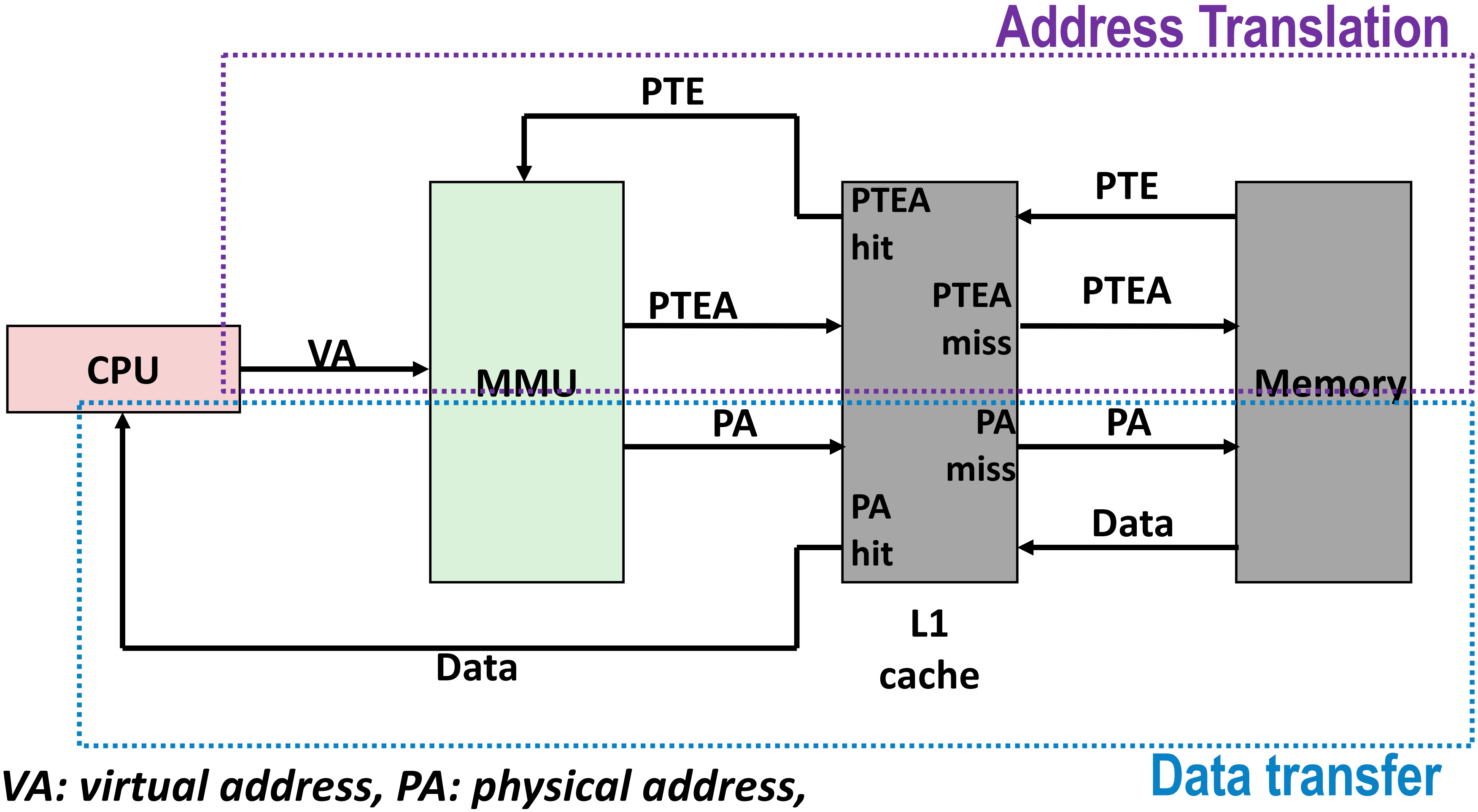
Digging Deeper: Integrating VM and Caching



Integrating VM and Cache: “The players”



Integrating VM and Cache: “The game”

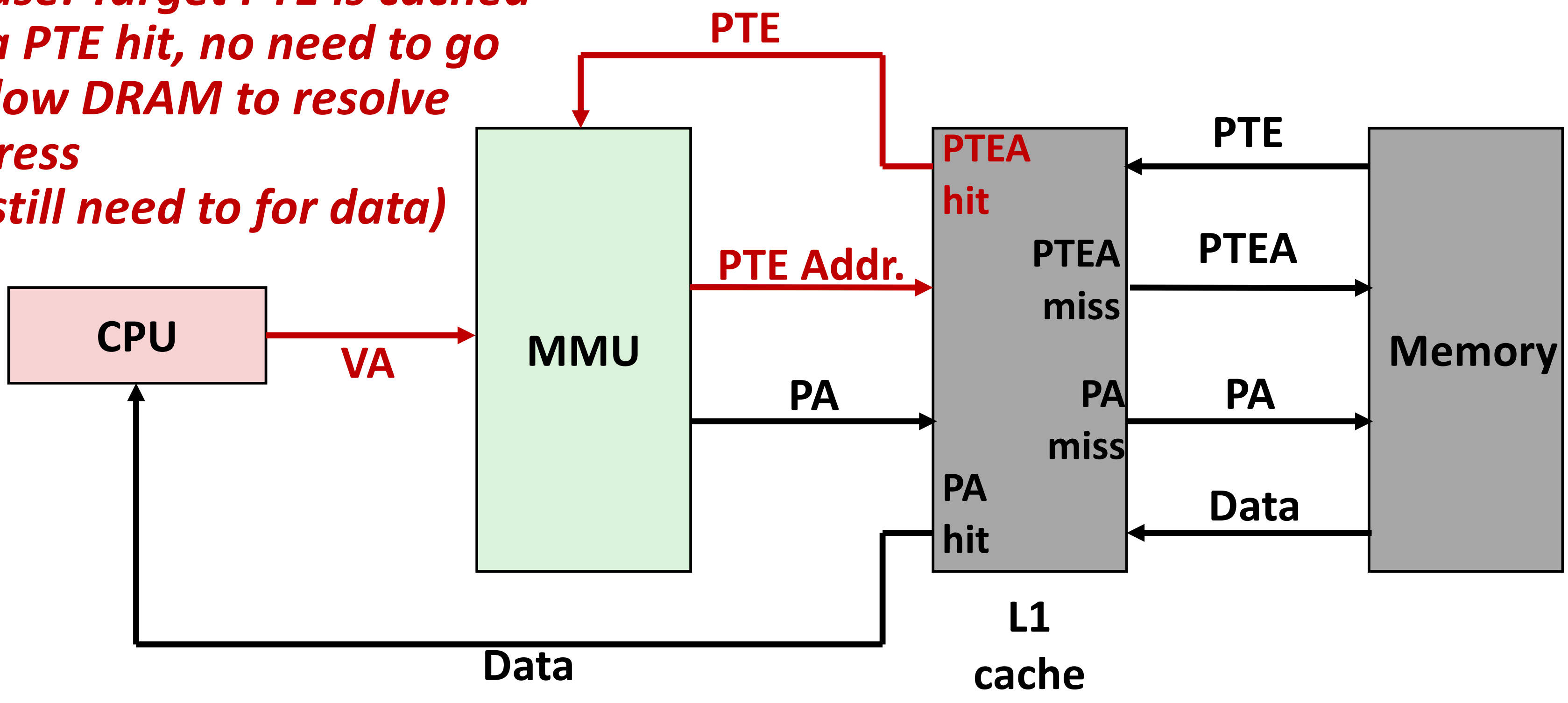


VA: virtual address, PA: physical address,
PTE: page table entry, PTEA = PTE address

Data transfer depends on address translation

Integrating VM and Cache

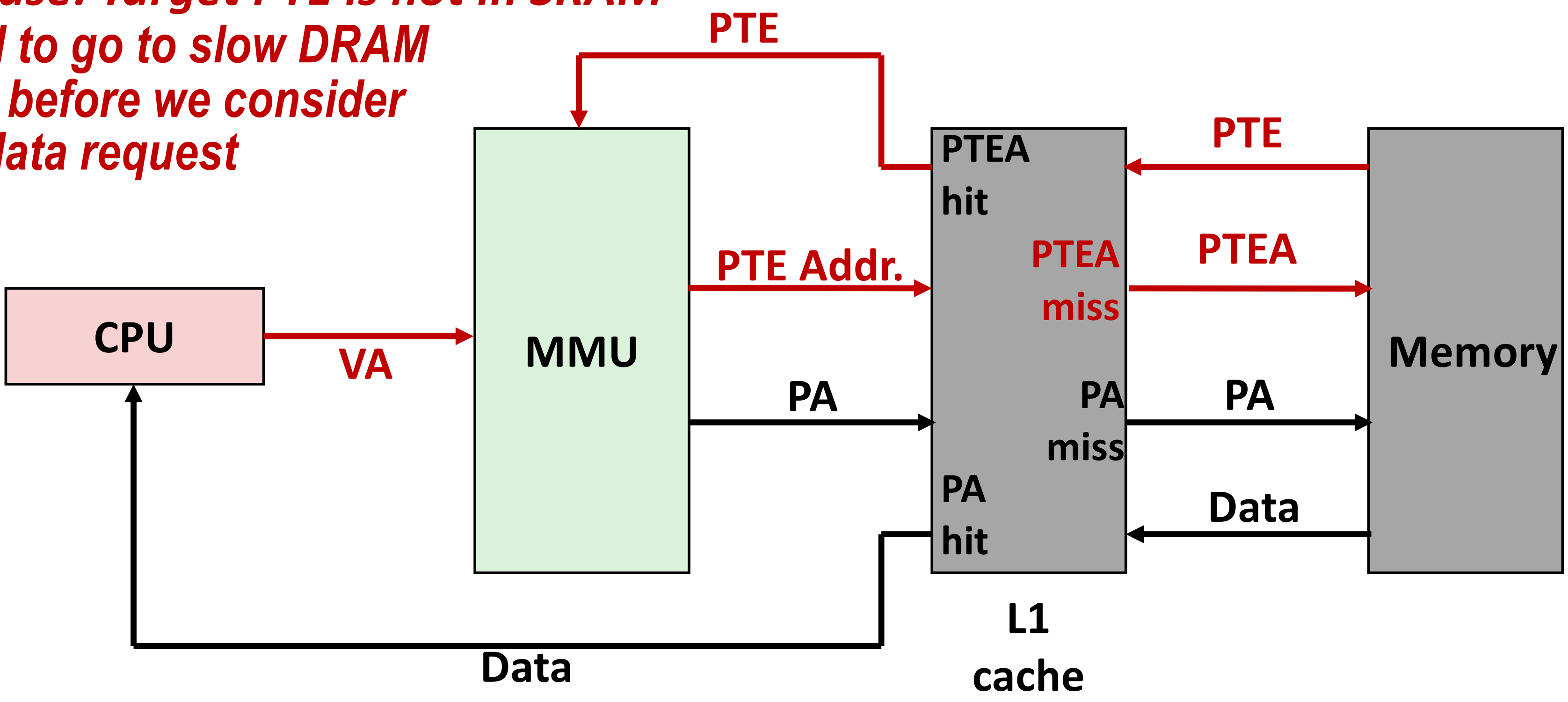
Fast case: Target PTE is cached
- on a PTE hit, no need to go to slow DRAM to resolve address
(may still need to for data)



VA: virtual address, PA: physical address,
PTE: page table entry, PTEA = PTE address

Integrating VM and Cache

Slow case: Target PTE is not in SRAM
- Need to go to slow DRAM even before we consider our data request



Integrating VM and Cache

Insight 1: Cache can hold page table entries, like any other data word!

Insight 2: Address translation happens BEFORE cache lookup.

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

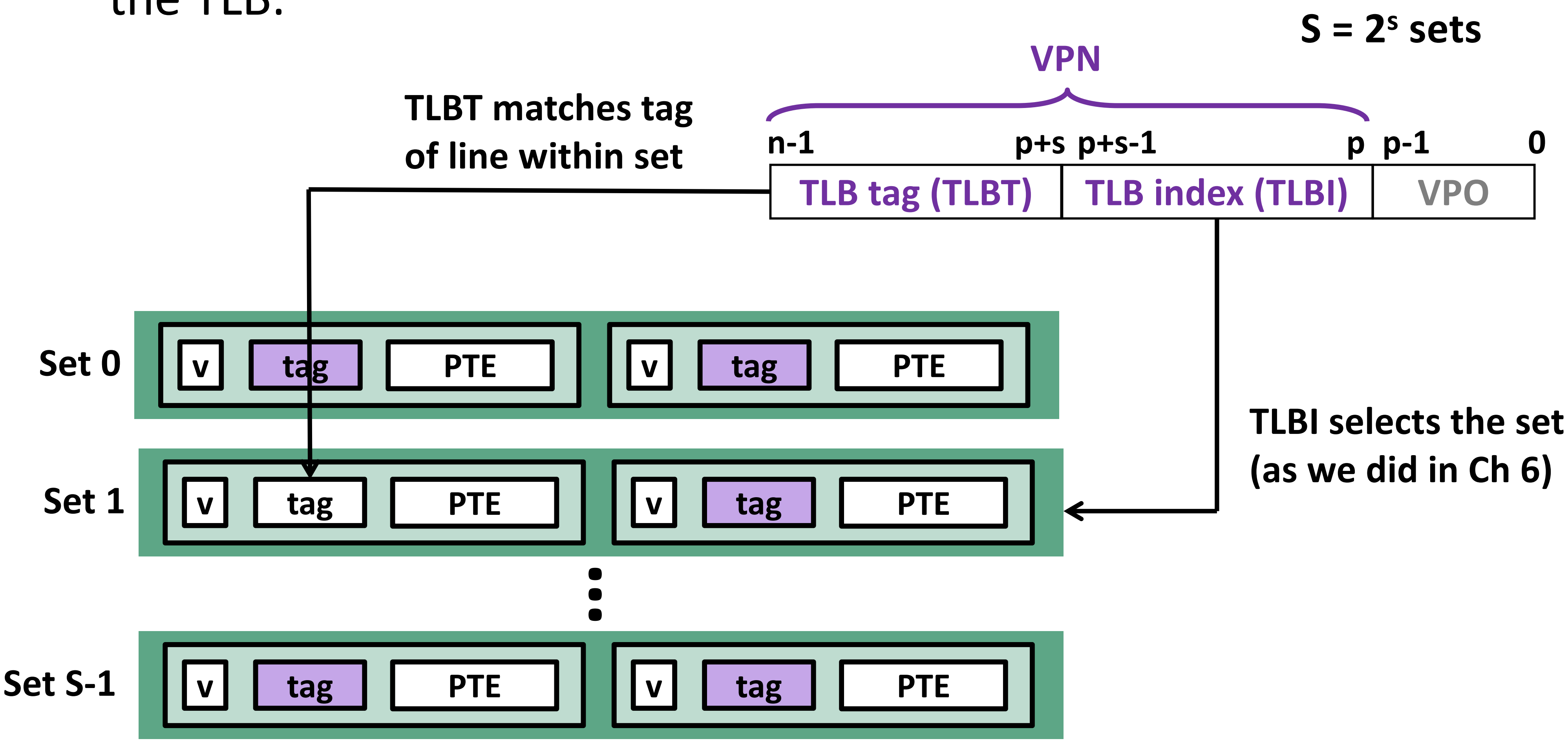
Speeding up Translation with a TLB

- **Problem**: Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - Even a PTE hit pays a small L1 delay
- **Solution**: *Translation Lookaside Buffer* (TLB)
  - Small set-associative **hardware cache** *in* MMU (part of CPU chip)
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for a small number of pages
  - Each cache line holds one block consisting of a single PTE

Similar to Instruction and Data cache separation.

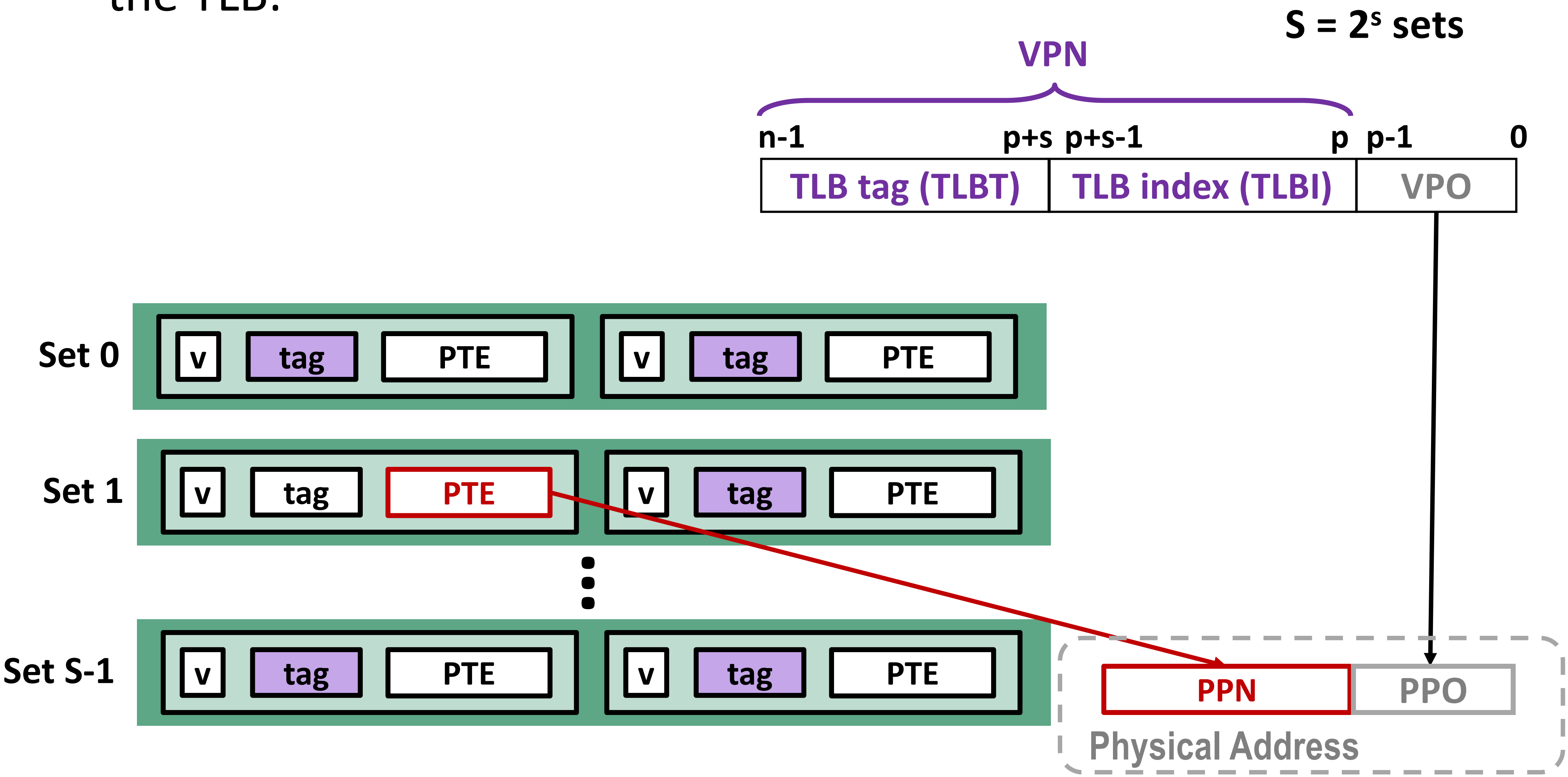
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

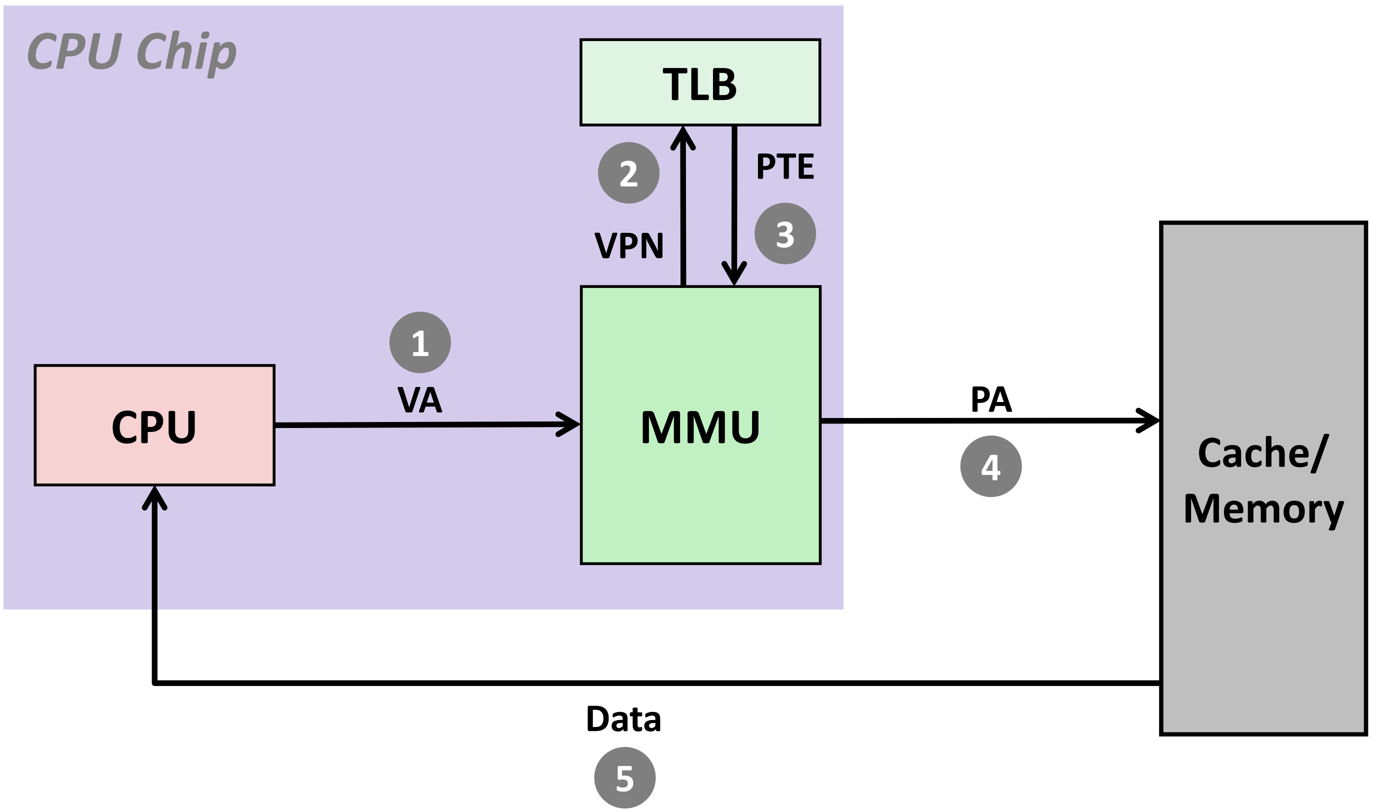


Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



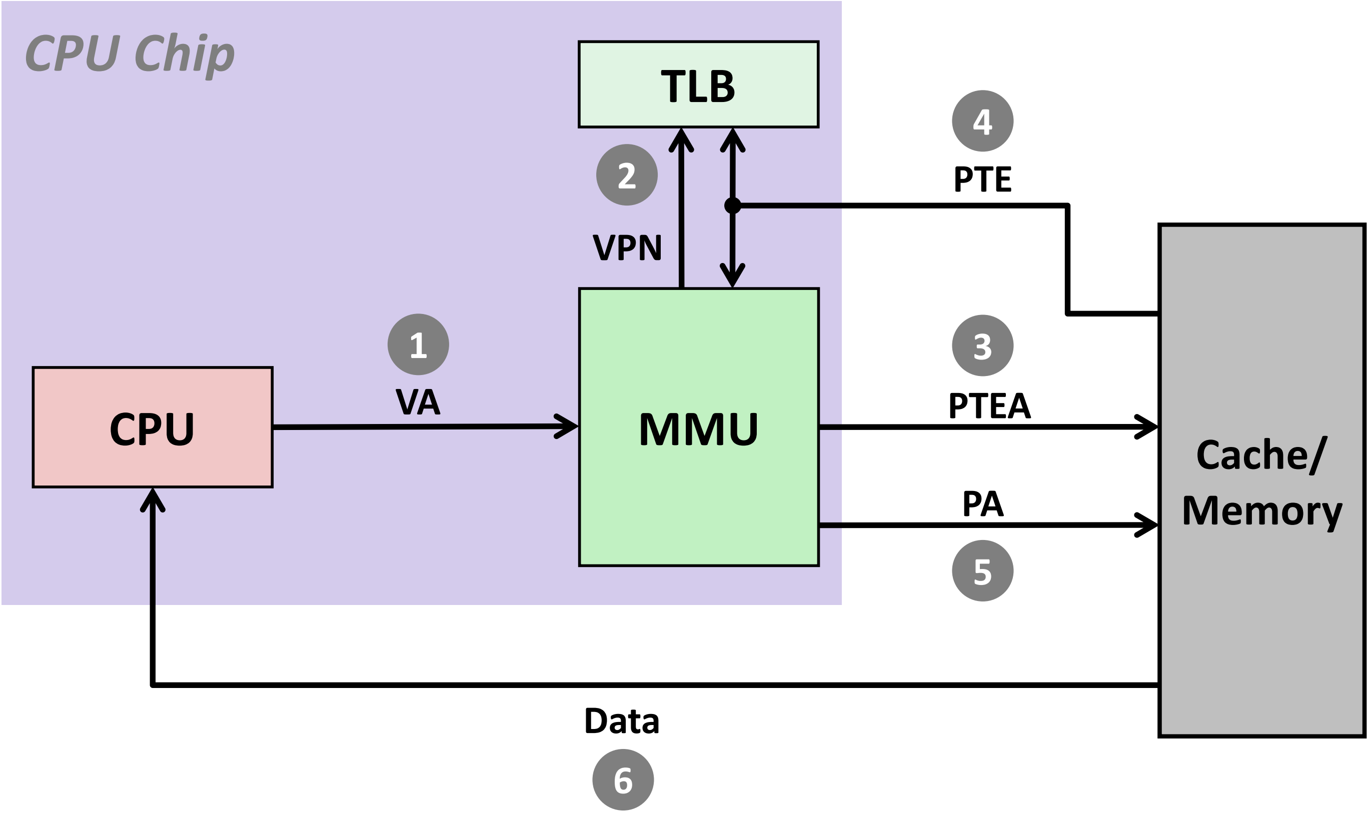
TLB Hit



A TLB hit eliminates a memory access.

All steps in address *translation* happen inside MMU and are fast.

TLB Miss



A TLB miss incurs an additional memory access (the PTE).

Fortunately, TLB misses are rare. Thanks, locality!

TLB Fun Facts

- May have separate instruction and data TLBs
- May have multiple levels of TLBs
- Who loads the TLB with entries?
  - Hardware-managed:
    - Page table walkers walk page table and update TLB
  - Software-managed TLB:
    - On TLB miss, OS walks page tables and loads TLB

Practice on Your Own

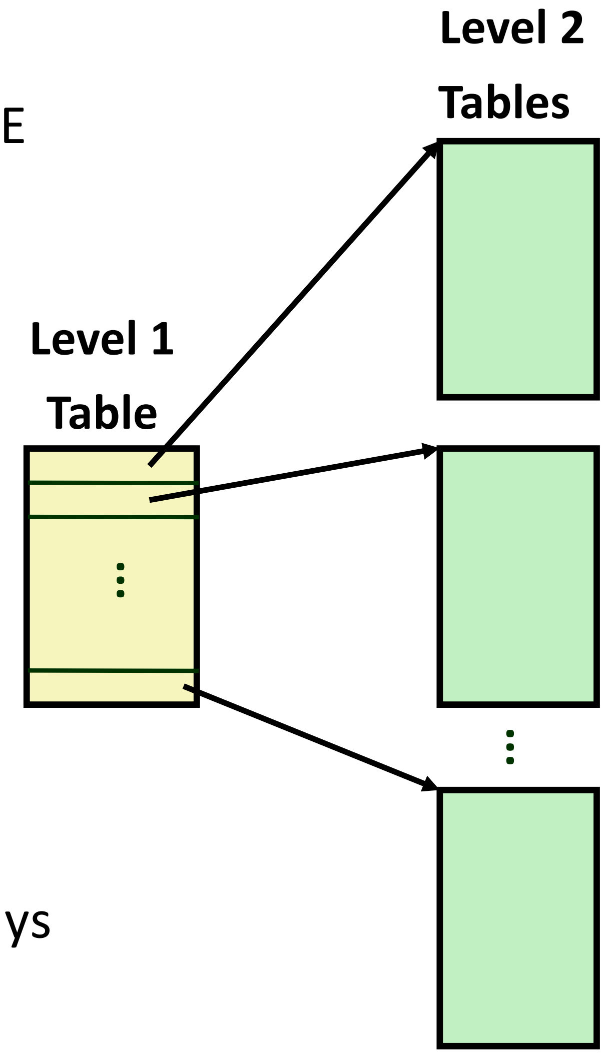
- How many times might a TLB need to be accessed when executing a single instruction (from fetch to write back)?

Page Table Structure

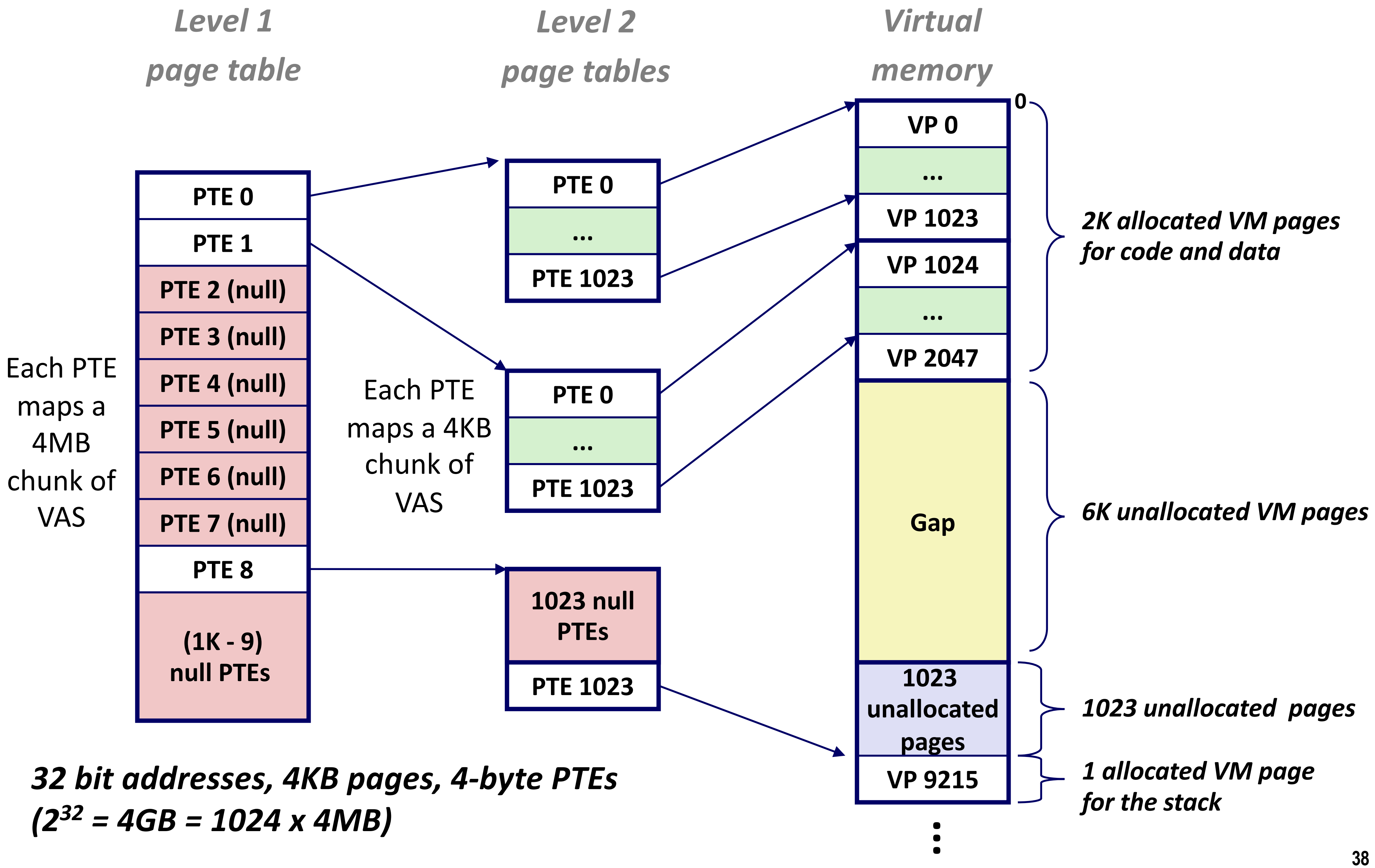
- Recall: A control register (CR3) holds the starting address of a process’s page table
  - How big is a process’s page table?
    - Size of a PTE (what is stored)?
    - Number of PTEs?
  - How big is a process’s working set (roughly speaking)?
    - Stack size?
    - Heap size?
    - Code/text?
- Observations:
  - Page table is HUGE, but sparsely populated
- What data structures might we use to represent our page table?

Multi-Level Page Tables: Concrete Example

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem?
  - Would need a 512 GB page table!
    - $2^{48} / 2^{12} = 2^{36}$ = # entries in page table
    - 2^{3} bytes per entry
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes in every page table
- Common solution: Multi-level page tables
  - No need to waste space for unallocated pages!
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)



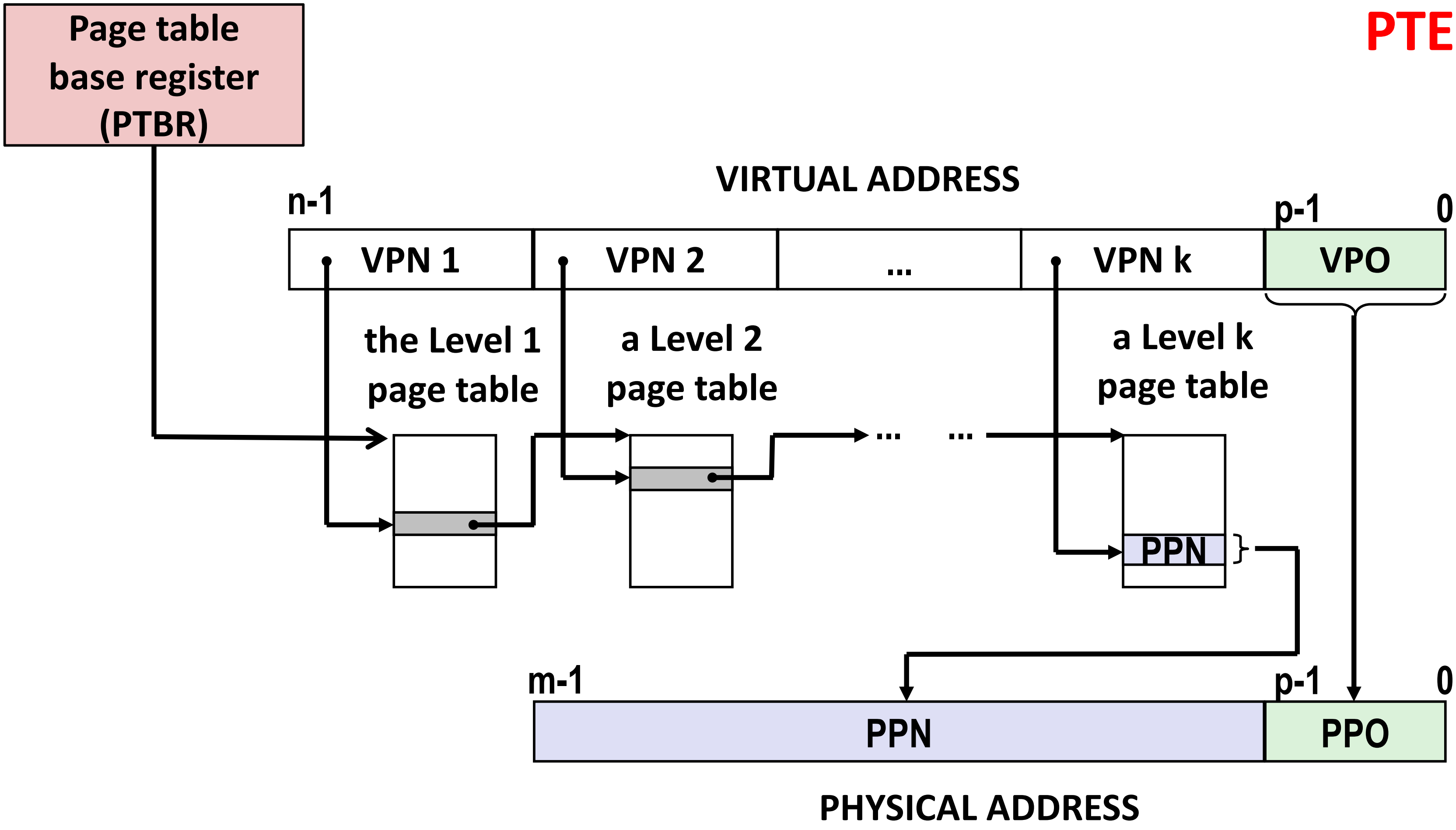
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs
(2^{32} = 4GB = 1024 x 4MB)

Translating with a k-level Page Table

TLB caches PTEs from all levels



Address Translation Summary (Ch 9.6)

- Programmer's view of virtual memory:
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- System's view of virtual memory:
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions