

# Digital Design & Computer Arch.

## Lecture 25b: Virtual Memory

Prof. Onur Mutlu

ETH Zürich  
Spring 2021  
3 June 2021

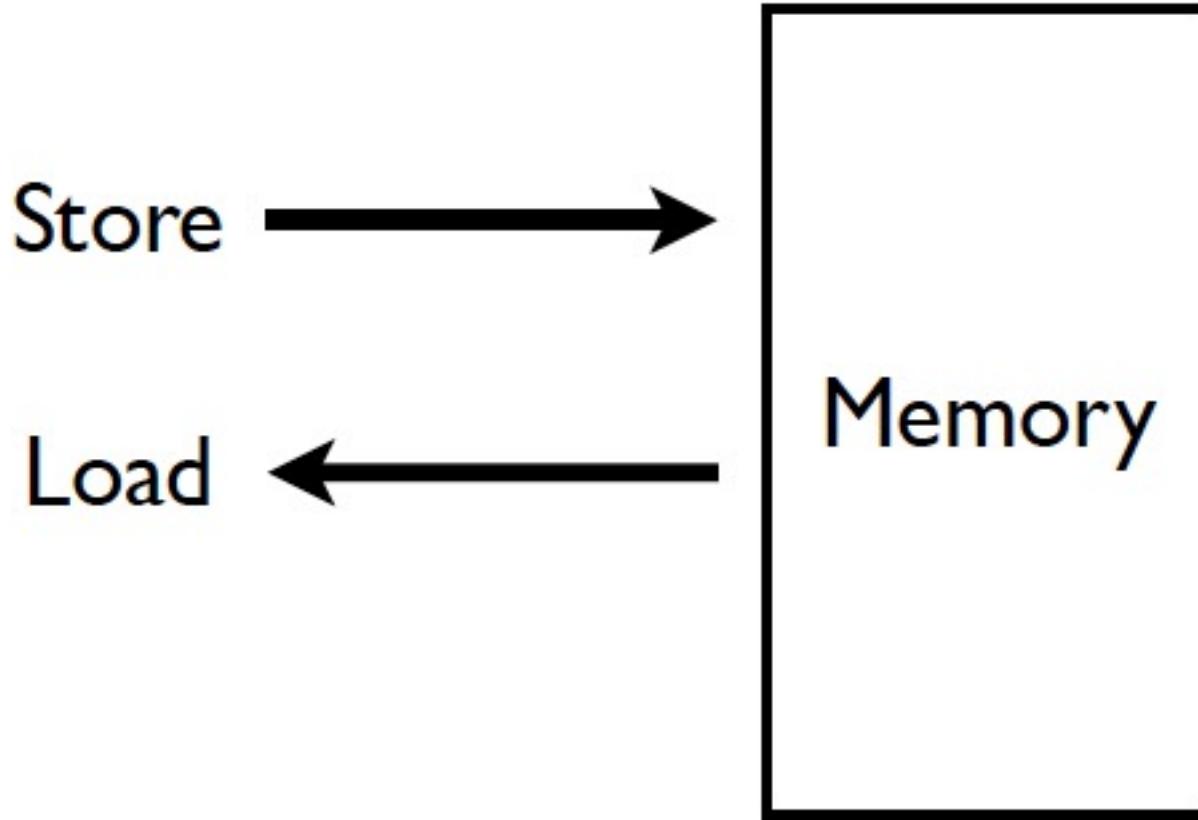
# Readings

---

- Virtual Memory
- Required
  - H&H Chapter 8.4
  - Kim & Mutlu, "Memory Systems," Computing Handbook, 2014.
    - [https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction\\_computing-handbook14.pdf](https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf)
- Recommended
  - Jacob & Mudge, "Virtual Memory: Issues of Implementation," IEEE Computer, 1998.
  - Hajinazar et al., "The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework," ISCA 2020.

# Memory (Programmer's View)

---



# Ideal Memory

---

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

# Abstraction: Virtual vs. Physical Memory

---

- Programmer sees virtual memory
  - Can assume the memory is “infinite”
- Reality: Physical memory size is much smaller than what the programmer assumes
- The system (system software + hardware, cooperatively) maps virtual memory addresses to physical memory
  - The system automatically manages the physical memory space transparently to the programmer
- + Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer
- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

---

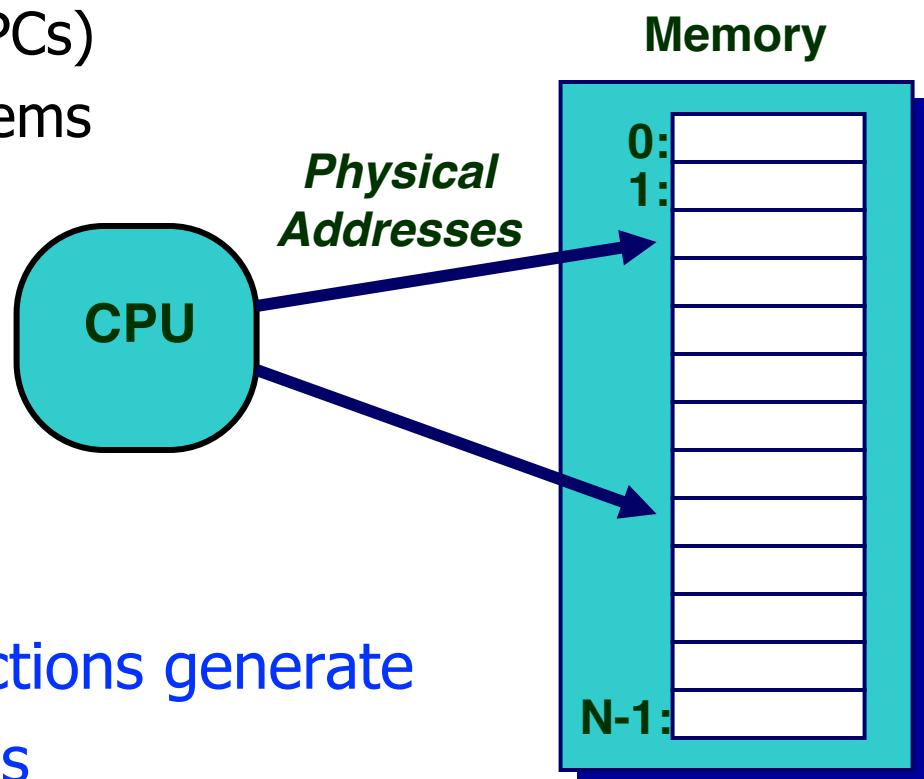
# Benefits of Automatic Management of Memory

---

- Programmer does not deal with physical addresses
- Each process has its own independent mapping of virtual→physical addresses
- Enables
  - Code and data to be located anywhere in physical memory  
(**relocation and flexible location of data**)
  - Isolation/separation of code and data of different processes in physical memory  
(**protection and isolation**)
  - Code and data sharing between multiple processes  
(**sharing**)

# A System with Physical Memory Only

- Examples:
  - ❑ most Cray supercomputers
  - ❑ early personal computers (PCs)
  - ❑ many older embedded systems



CPU's **load or store** instructions generate  
**physical** memory addresses

# The Problem

---

- Physical memory is of limited size (cost)
    - What if you need more?
    - Should the programmer be concerned about the size of code/data blocks fitting physical memory?
    - Should the programmer manage data movement from disk to physical memory?
  - Multiple programs may need the physical memory
    - Should the programmer make sure all processes (different programs) can fit in physical memory?
    - Should the programmer ensure two processes do not unintentionally or incorrectly use the same physical memory portion?
  - ISA can have an address space greater than the physical memory size
    - E.g., a 64-bit address space with byte addressability
    - What if you do not have enough physical memory?
-

# Difficulties of Direct Physical Addressing

---

- Programmer needs to manage physical memory space
  - Inconvenient & difficult
  - More difficult when you have **multiple processes**
- Difficult to support code and data relocation
  - Addresses are directly specified in the program
- Difficult to support multiple processes
  - Protection and isolation between multiple processes
  - Sharing of physical memory space without problems
- Difficult to support data/code sharing across processes
  - Different processes need to reference the same physical address

# Virtual Memory

---

- Idea: Give each program the illusion of a large address space while having a small physical memory
  - So that the programmer does not worry about managing physical memory (within a process or across processes)
- Programmer can assume they have “infinite” amount of physical memory
- Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  - Illusion is maintained for each independent process

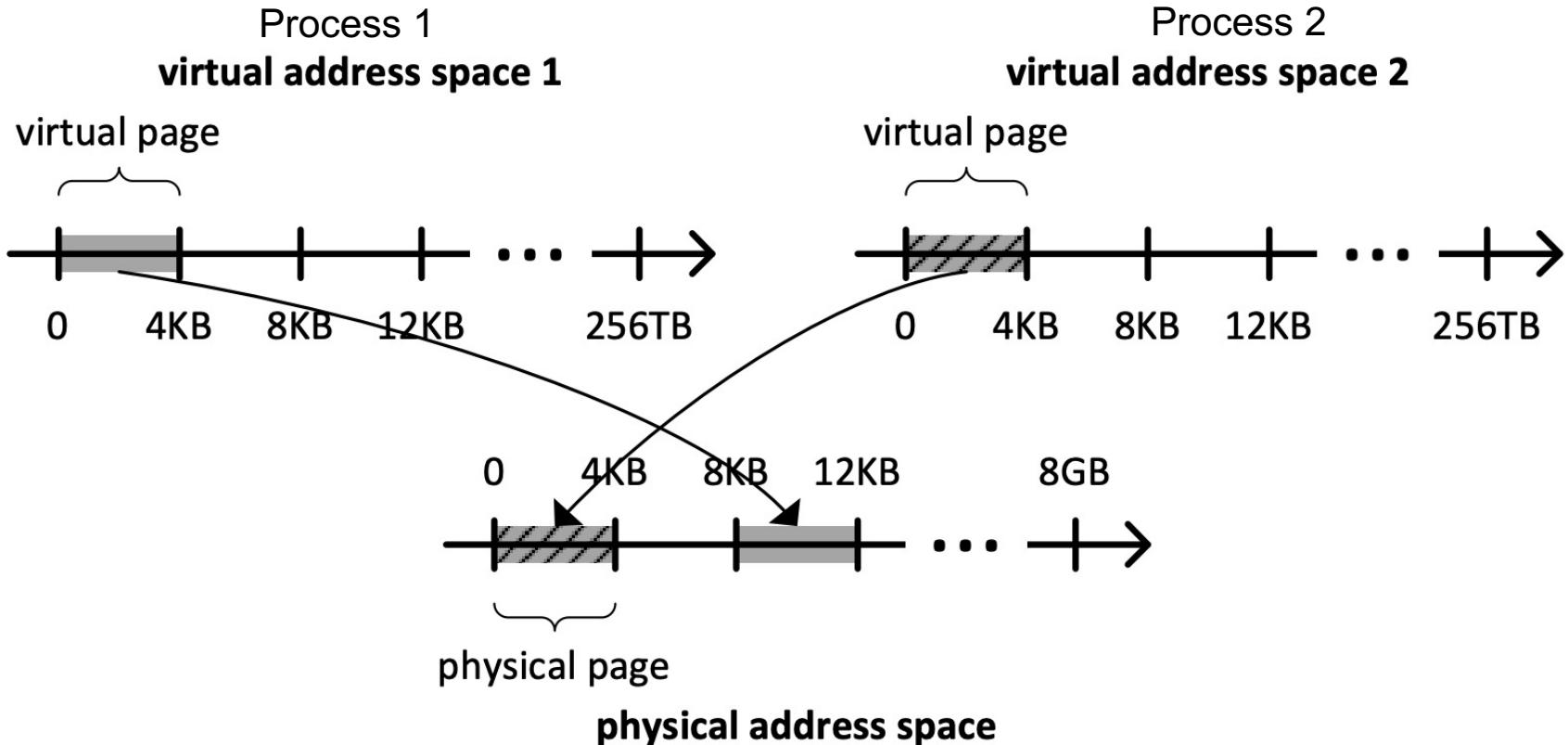
# Basic Mechanism

---

- Indirection (in addressing)
- Address generated by each instruction in a program is a “virtual address”
  - i.e., it is not the physical address used to address main memory
  - called “linear address” in x86
- An “address translation” mechanism maps this address to a “physical address”
  - called “real address” in x86
  - Address translation mechanism can be implemented in hardware and software together

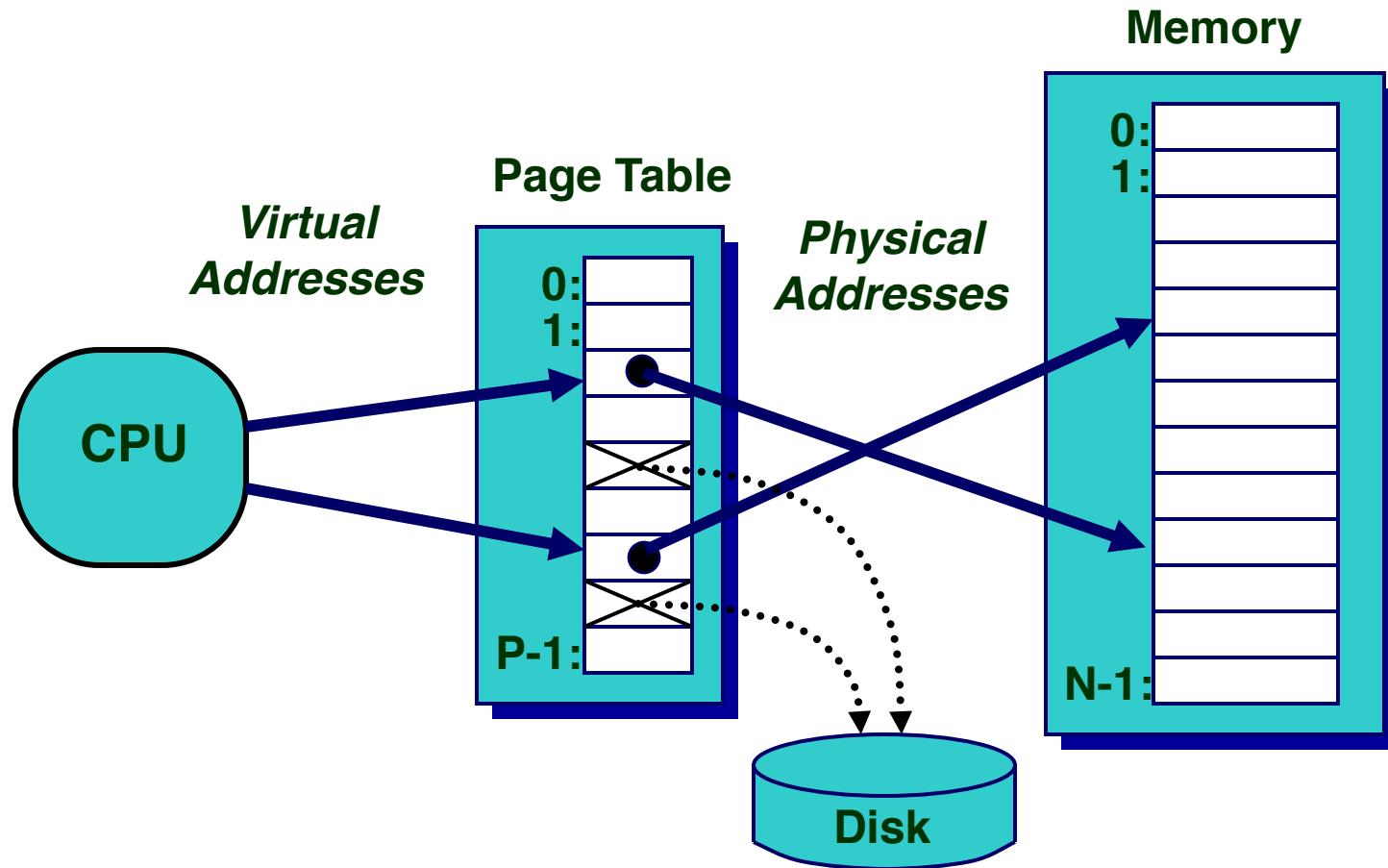
# Virtual Memory: Conceptual View

## ■ Illusion of large, separate address space per process



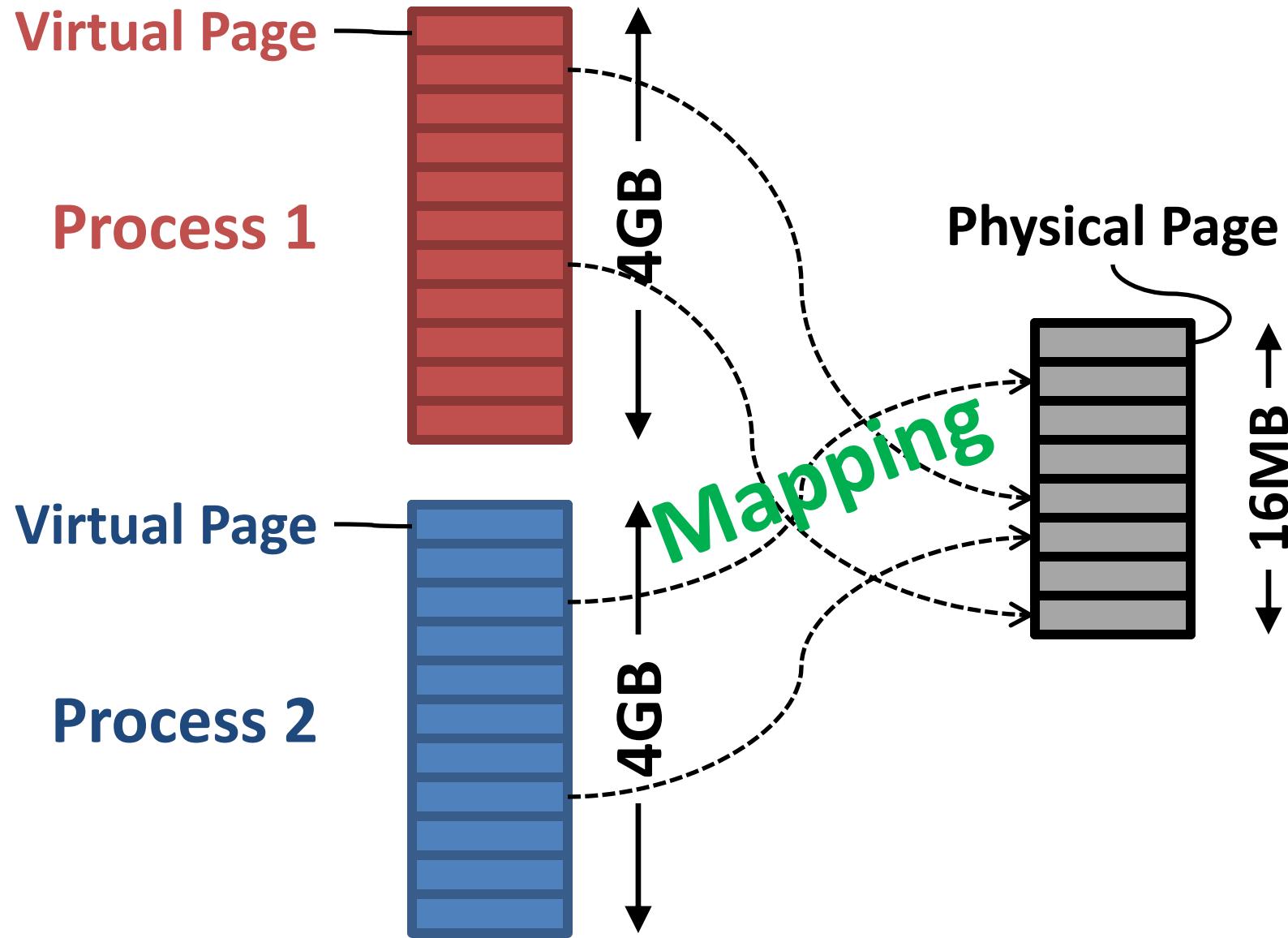
Requires **indirection and mapping** between virtual and physical spaces

# A System with Virtual Memory (Page-based)



- **Address Translation:** The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

# Page-based Virtual-to-Physical Mapping



# Four Issues in Indirection and Mapping

---

- When to map a virtual address to a physical address?
  - When the virtual address is first referenced by the program
- What is the mapping granularity?
  - Byte? Kilo-byte? Mega-byte? ...
  - Multiple granularities?
- Where and how to store the virtual→physical mappings?
  - Operating system data structures? Hardware? Cooperative?
- What to do when physical address space is full?
  - Evict an unlikely-to-be-needed virtual address from physical memory

# Virtual Pages, Physical Frames

---

- Virtual address space divided into **pages**
  - Physical address space divided into **frames**
  
  - A virtual page is mapped to
    - A physical frame, if the page is in physical memory
    - A location in disk, otherwise
  
  - If an accessed virtual page is not in memory, but on disk
    - Virtual memory system brings the page into a physical frame and adjusts the mapping → this is called **demand paging**
  
  - **Page table** is the table that stores the mapping of virtual pages to physical frames
-

# Physical Memory as a Cache

---

- In other words...
- Physical memory is a cache for pages stored on disk
  - In fact, it is a fully-associative cache in modern systems (a virtual page can potentially be mapped to any physical frame)
- Similar caching issues exist as we have covered earlier:
  - Placement: where and how to place/find a page in cache?
  - Replacement: what page to remove to make room in cache?
  - Granularity of management: large, small, uniform pages?
  - Write policy: what do we do about writes? Write back?

# Cache/Virtual Memory Analogues

---

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Index/Tag	Virtual Page Number

# Virtual Memory Definitions

---

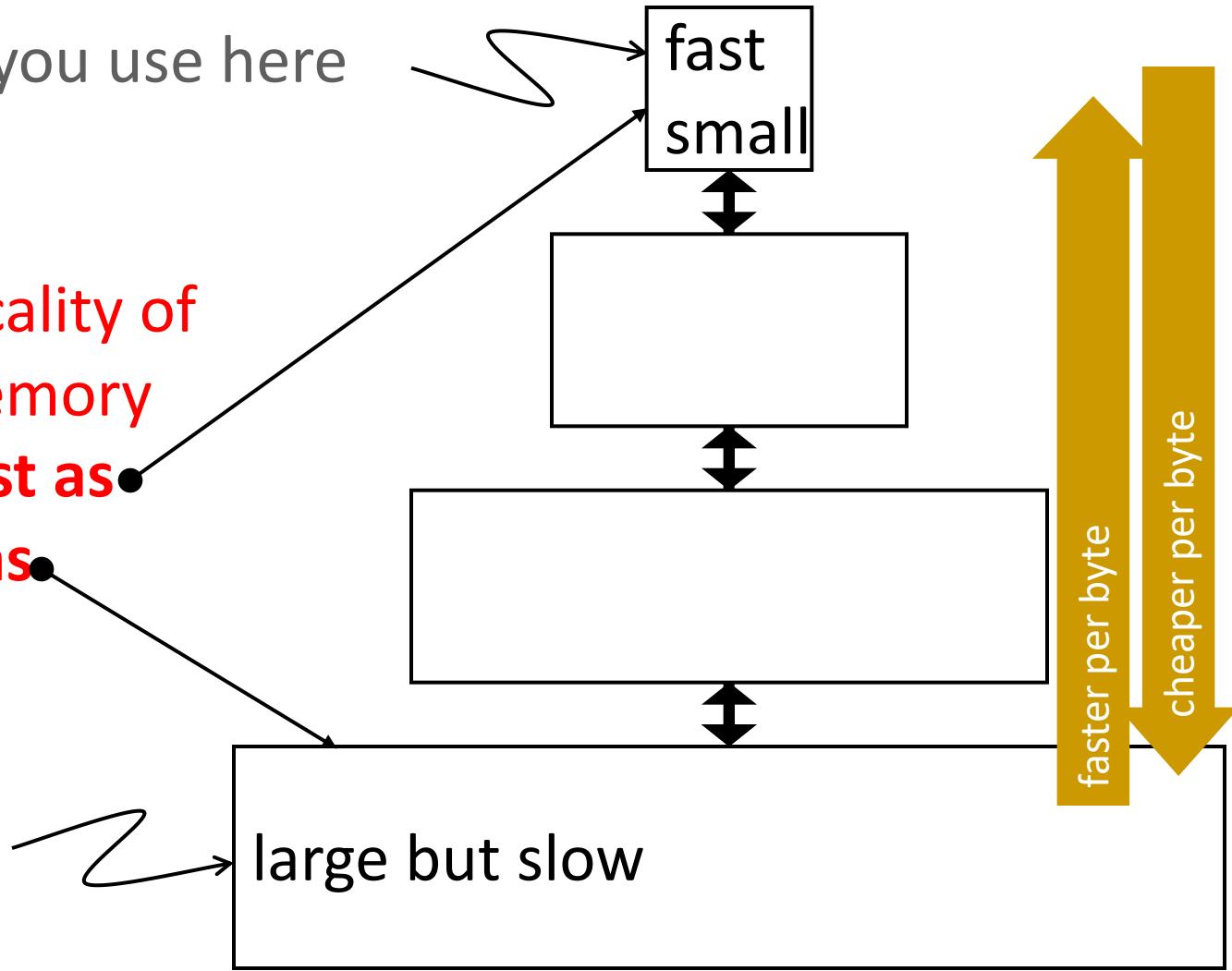
- **Page size:** the mapping granularity of virtual→physical address spaces
    - dictates the amount of data transferred from hard disk to DRAM at once
  - **Page table:** table that stores virtual→physical page mappings
    - lookup table used to translate virtual page addresses to physical frame addresses (and find where the associated data is)
  - **Address translation:** the process of determining the physical address from the virtual address
-

# Recall: The Memory Hierarchy

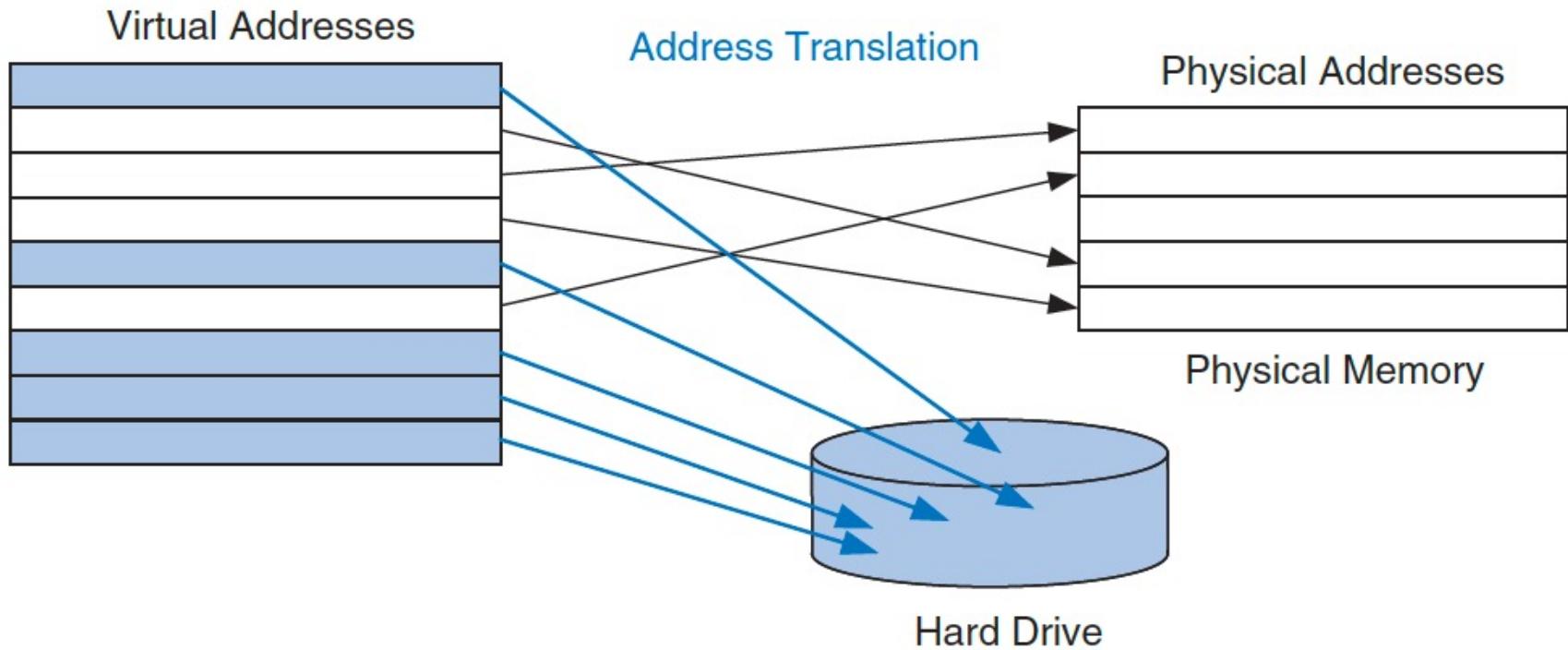
move what you use here

With good locality of reference, memory appears **as fast as** and **as large as**

backup everything here



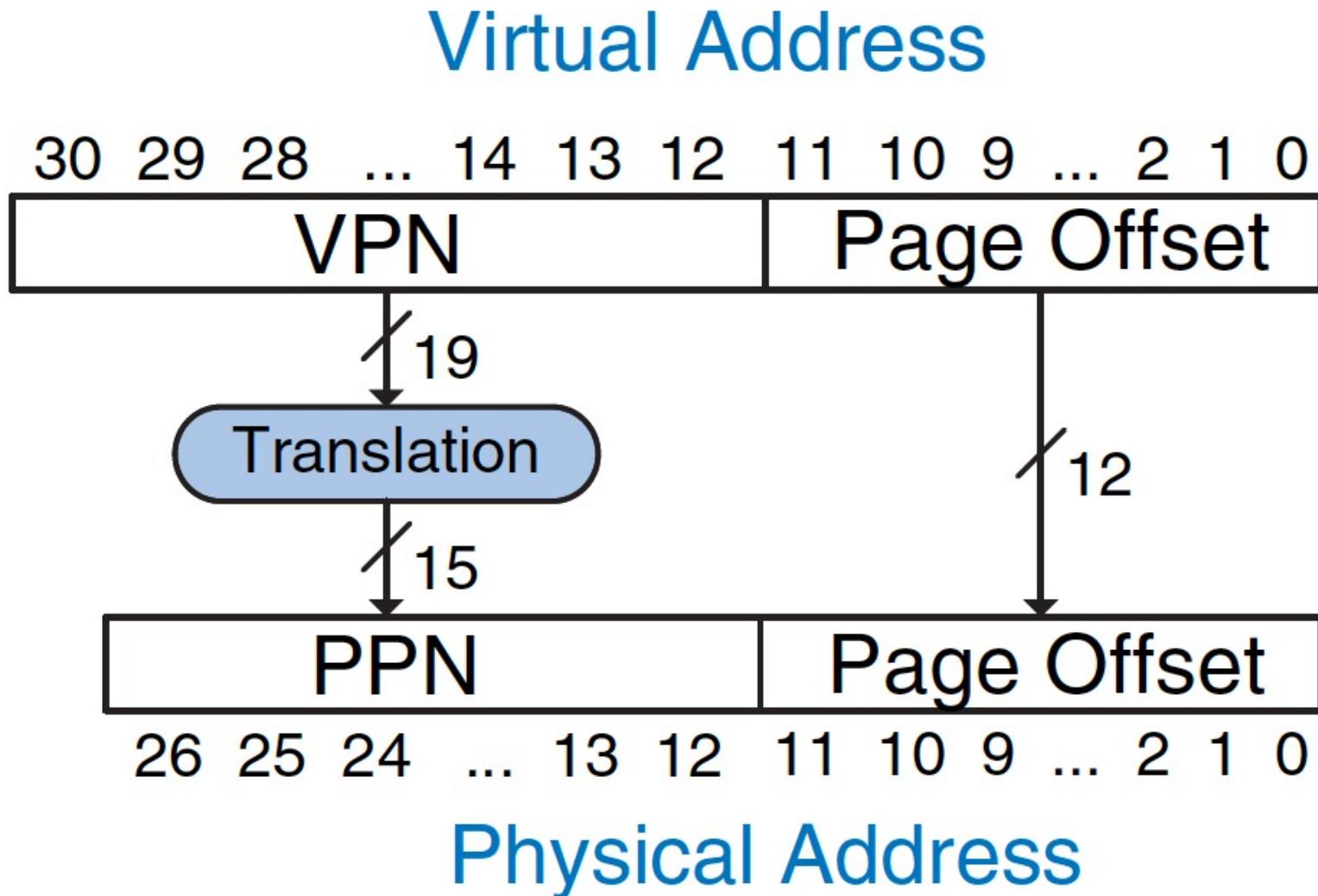
# Virtual to Physical Mapping



- Most accesses hit in physical memory
- Programs see the large capacity of virtual memory

# Address Translation

---



# Virtual Memory Example

---

## ■ System:

- Virtual memory size:  $2 \text{ GB} = 2^{31} \text{ bytes}$
- Physical memory size:  $128 \text{ MB} = 2^{27} \text{ bytes}$
- Page size:  $4 \text{ KB} = 2^{12} \text{ bytes}$

# Virtual Memory Example (Continued)

---

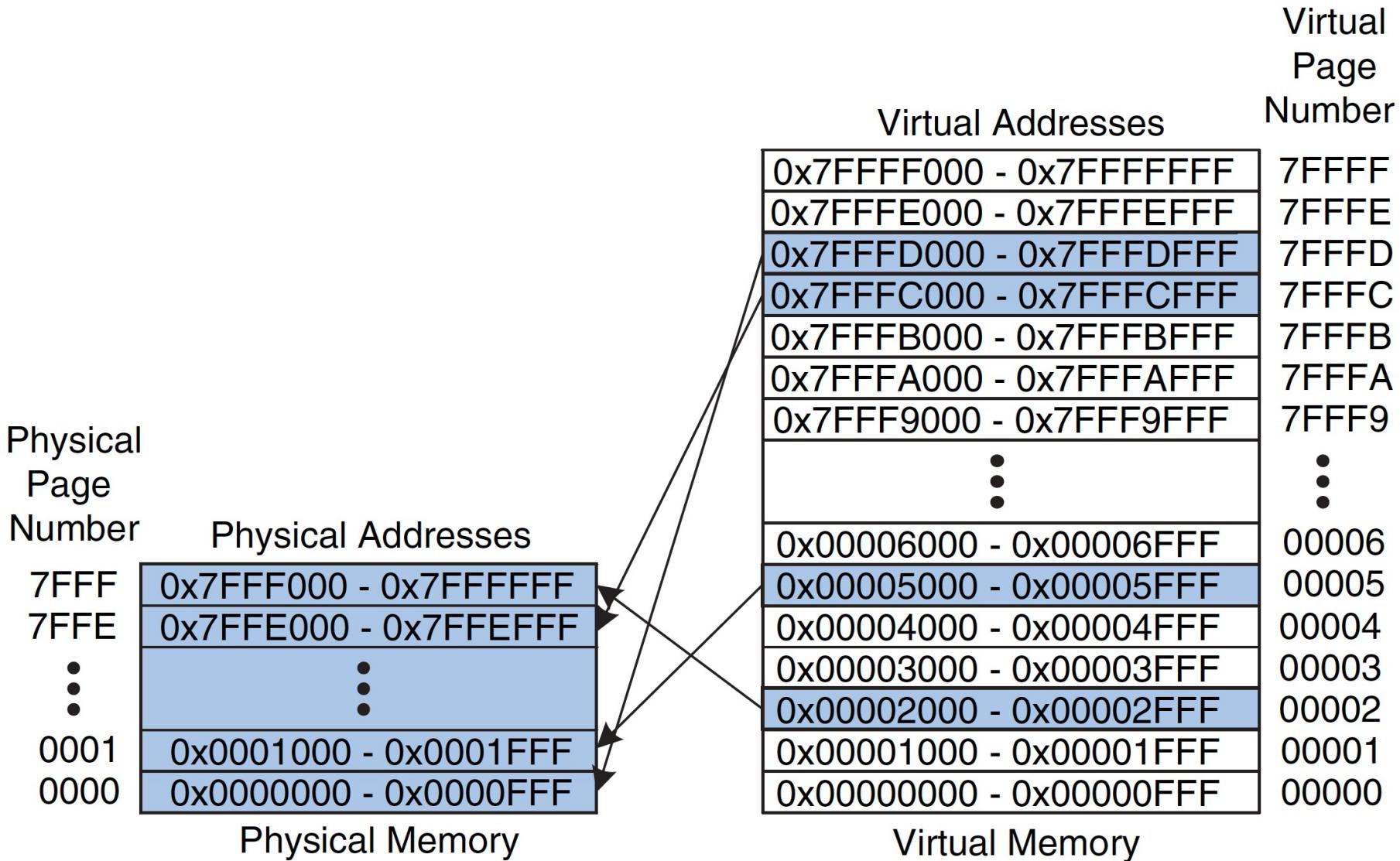
## ■ System:

- Virtual memory size:  $2 \text{ GB} = 2^{31}$  bytes
- Physical memory size:  $128 \text{ MB} = 2^{27}$  bytes
- Page size:  $4 \text{ KB} = 2^{12}$  bytes

## ■ Organization:

- Virtual address: **31** bits
- Physical address: **27** bits
- Page offset: **12** bits
- # Virtual pages =  $2^{31}/2^{12} = \mathbf{2^{19}}$  (VPN = 19 bits)
- # Physical pages =  $2^{27}/2^{12} = \mathbf{2^{15}}$  (PPN = 15 bits)

# Virtual Memory Example (Continued)



# How Do We Translate Addresses?

---

## ■ **Page table**

- Has entry for each virtual page

## ■ Each **page table entry** has:

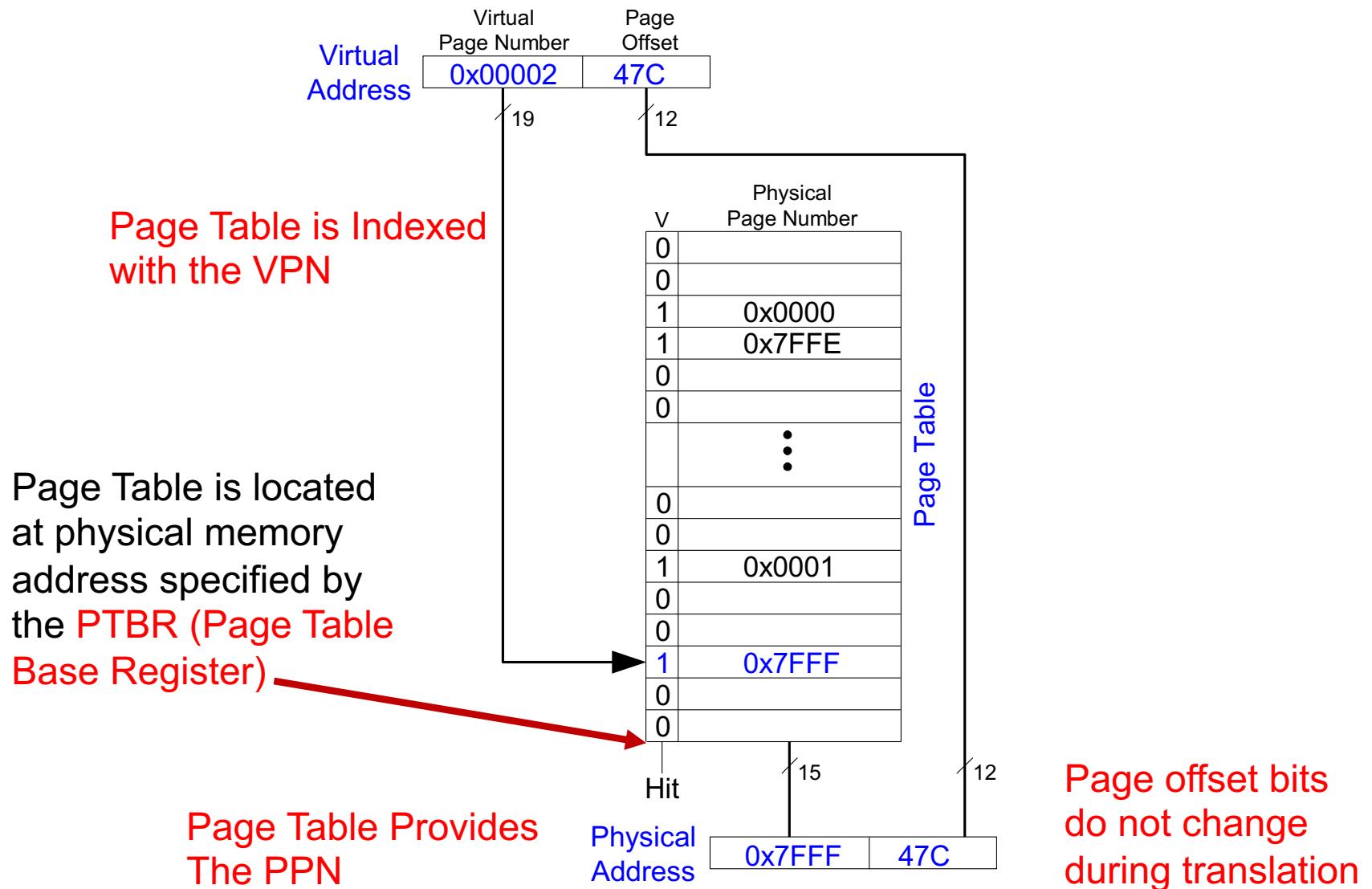
- **Valid bit**: whether the virtual page is located in physical memory (if not, it must be fetched from the hard disk)
- **Physical page number**: where the virtual page is located in physical memory
- (Replacement policy, dirty/modified, permission/access bits)

# Page Table for Our Example (Continued)

	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

# Page Table Address Translation Example



# Page Table Address Translation Example 1

- What is the physical address of virtual address 0x5F20?
- We first need to **find the page table entry containing the translation** for the corresponding VPN
- Look up the PTE at the address
  - $\text{PTBR} + \text{VPN} * \text{PTE-size}$

Page Table

v	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
⋮	
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

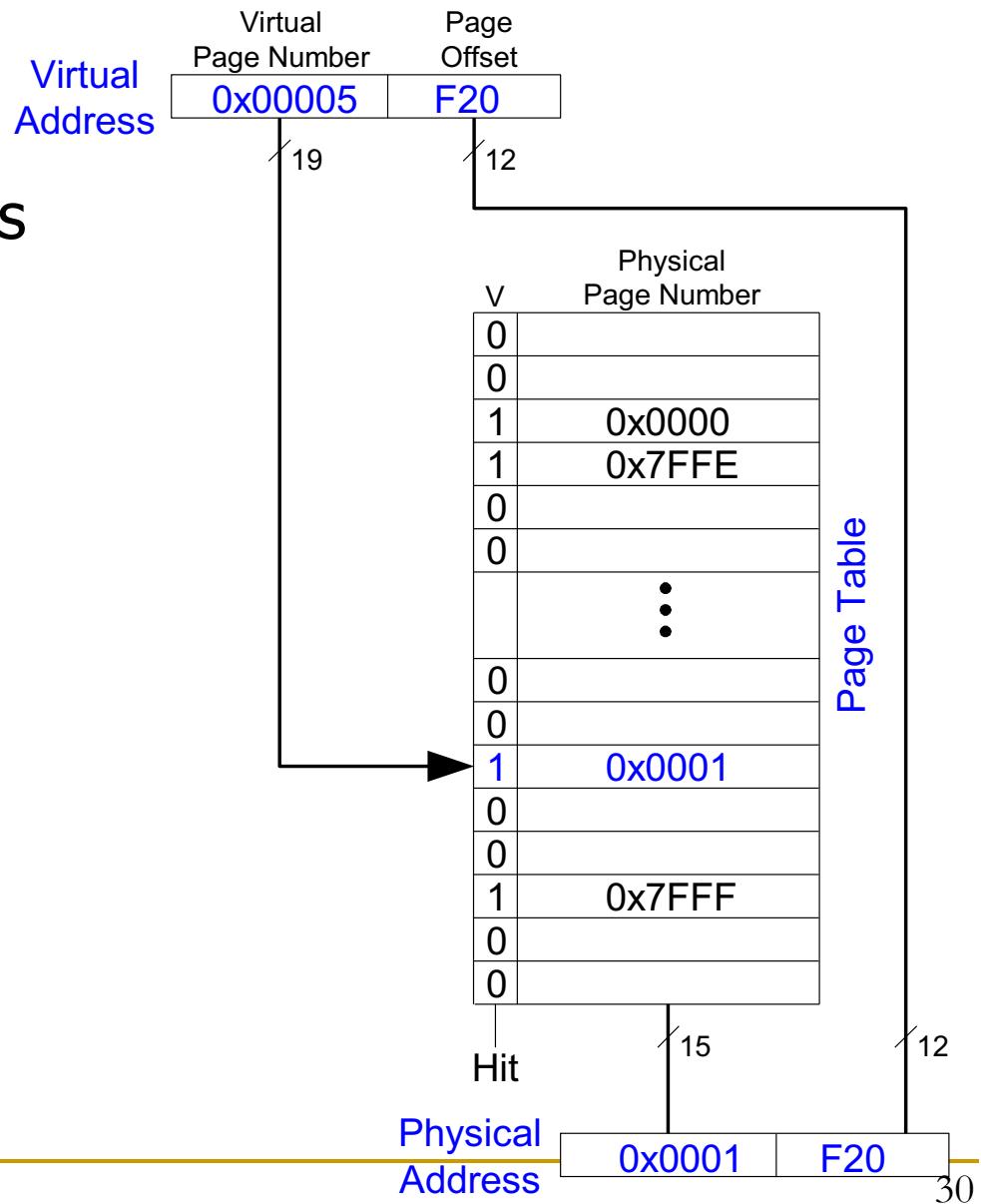
15

Hit

# Page Table Address Translation Example 1

- What is the physical address of virtual address 0x5F20?

- VPN = 5
- Entry 5 in page table indicates VPN 5 is in physical page 1
- Physical address is 0x1F20



# Page Table Address Translation Example 2

- What is the physical address of virtual address 0x73E0?

V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
⋮	
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

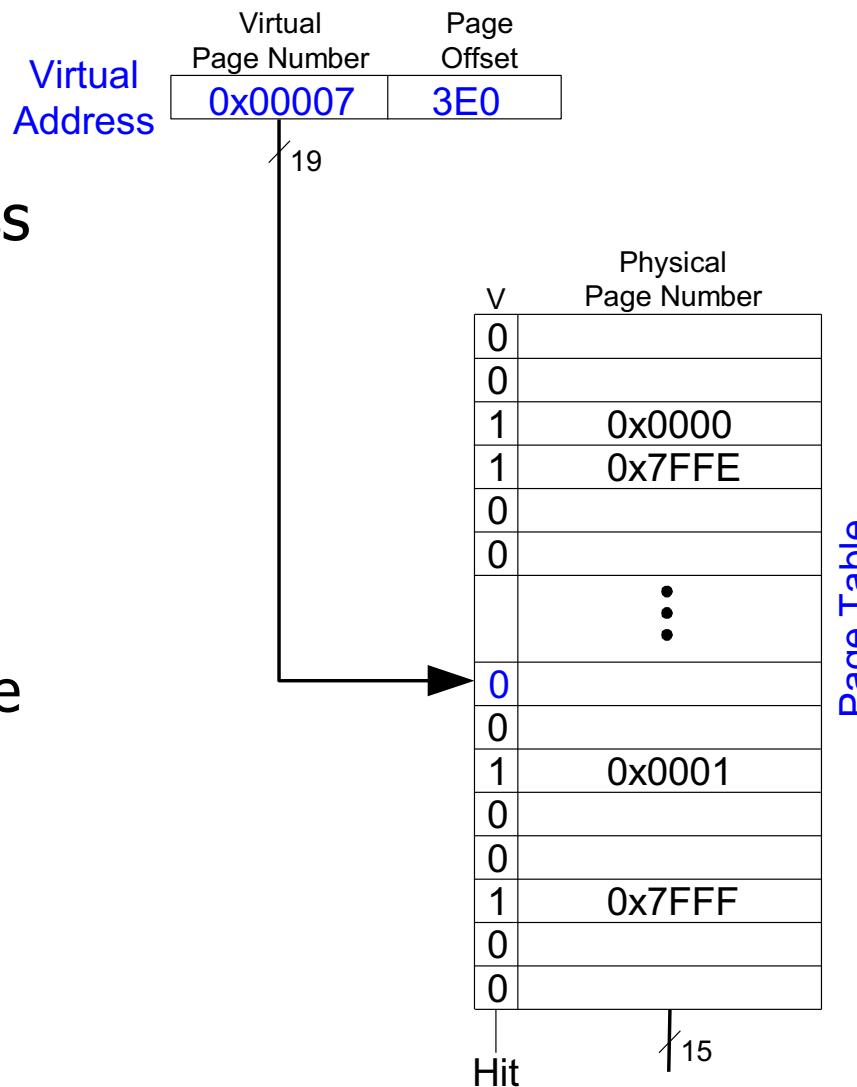
Page Table

15  
Hit

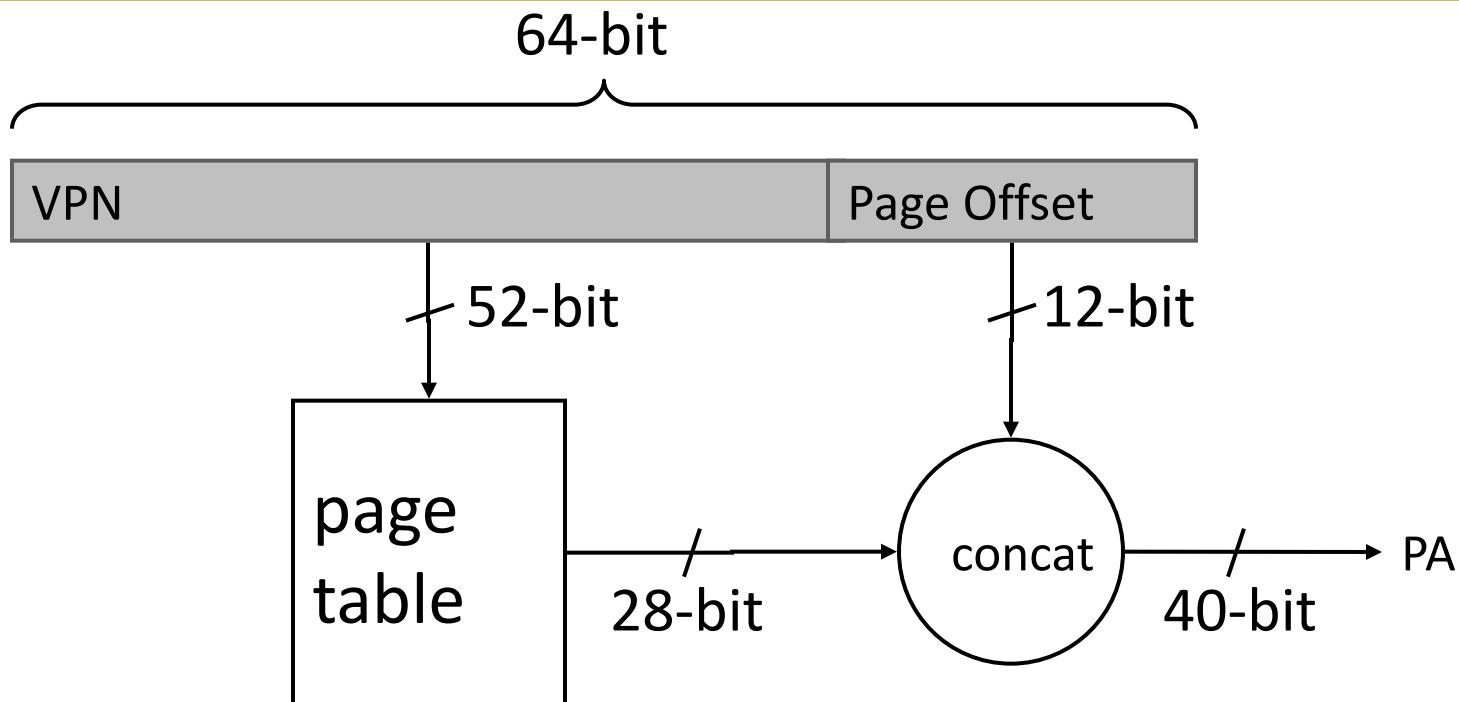
# Page Table Address Translation Example 2

- What is the physical address of virtual address 0x73E0?

- VPN = 7
- Entry 7 in page table is invalid, so the page is not in physical memory
- The virtual page must be swapped into physical memory from disk



# Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  - **$2^{52}$  entries x ~4 bytes  $\approx 2^{54}$  bytes**  
and that is for just one process!  
and the process may not be using the entire VM space!

# Page Table Challenges (I)

---

- Challenge 1: **Page table is large**
  - at least part of it needs to be located in physical memory
  - solution: multi-level (hierarchical) page tables

# Digital Design & Computer Arch.

## Lecture 25b: Virtual Memory

Prof. Onur Mutlu

ETH Zürich  
Spring 2021  
3 June 2021

We Will Cover the Following  
Slides in the Next Lecture

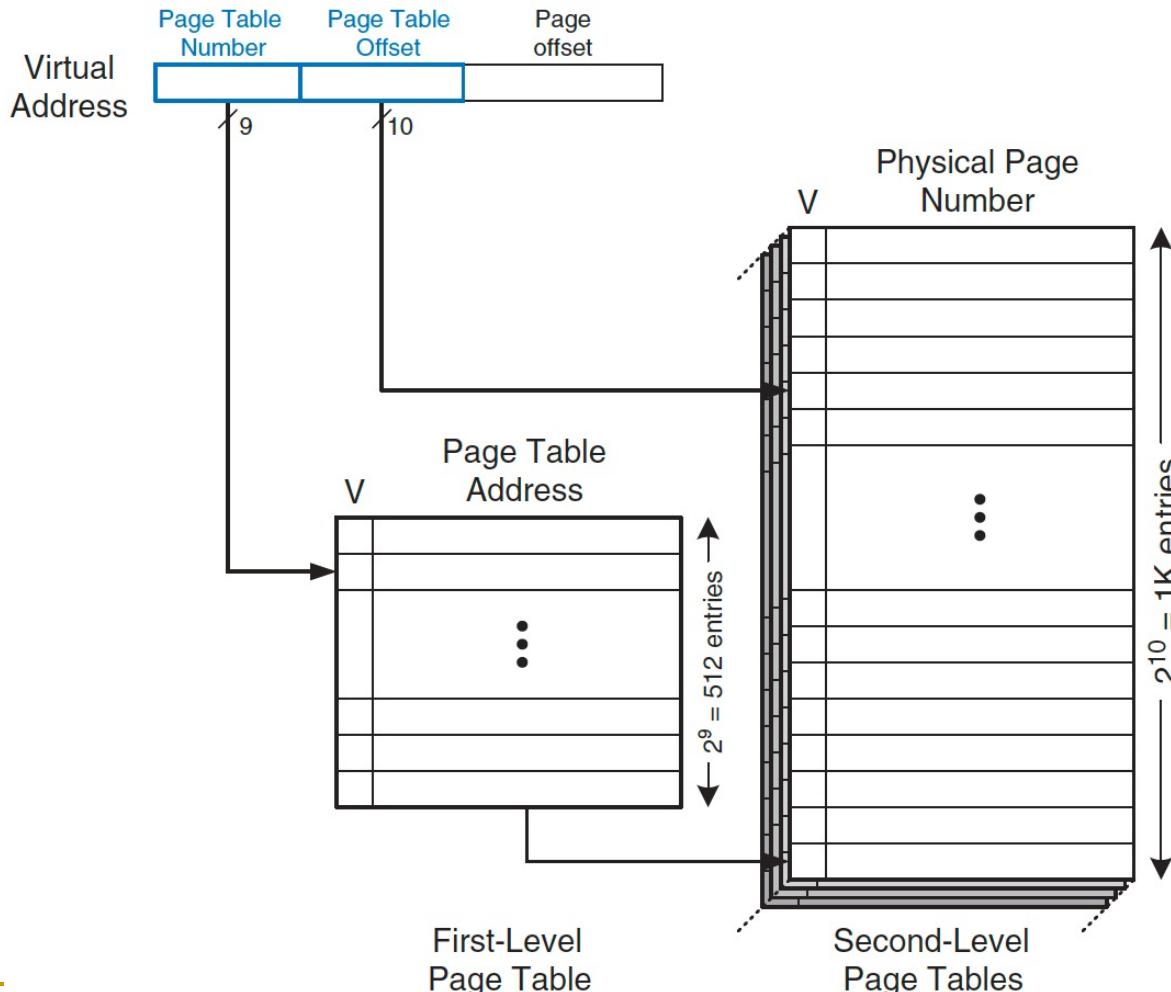
# Multi-Level Page Tables

---

- Idea: Organize page table in a hierarchical manner such that only a small first-level page table has to be in physical memory
- Multi-level (hierarchical) page tables

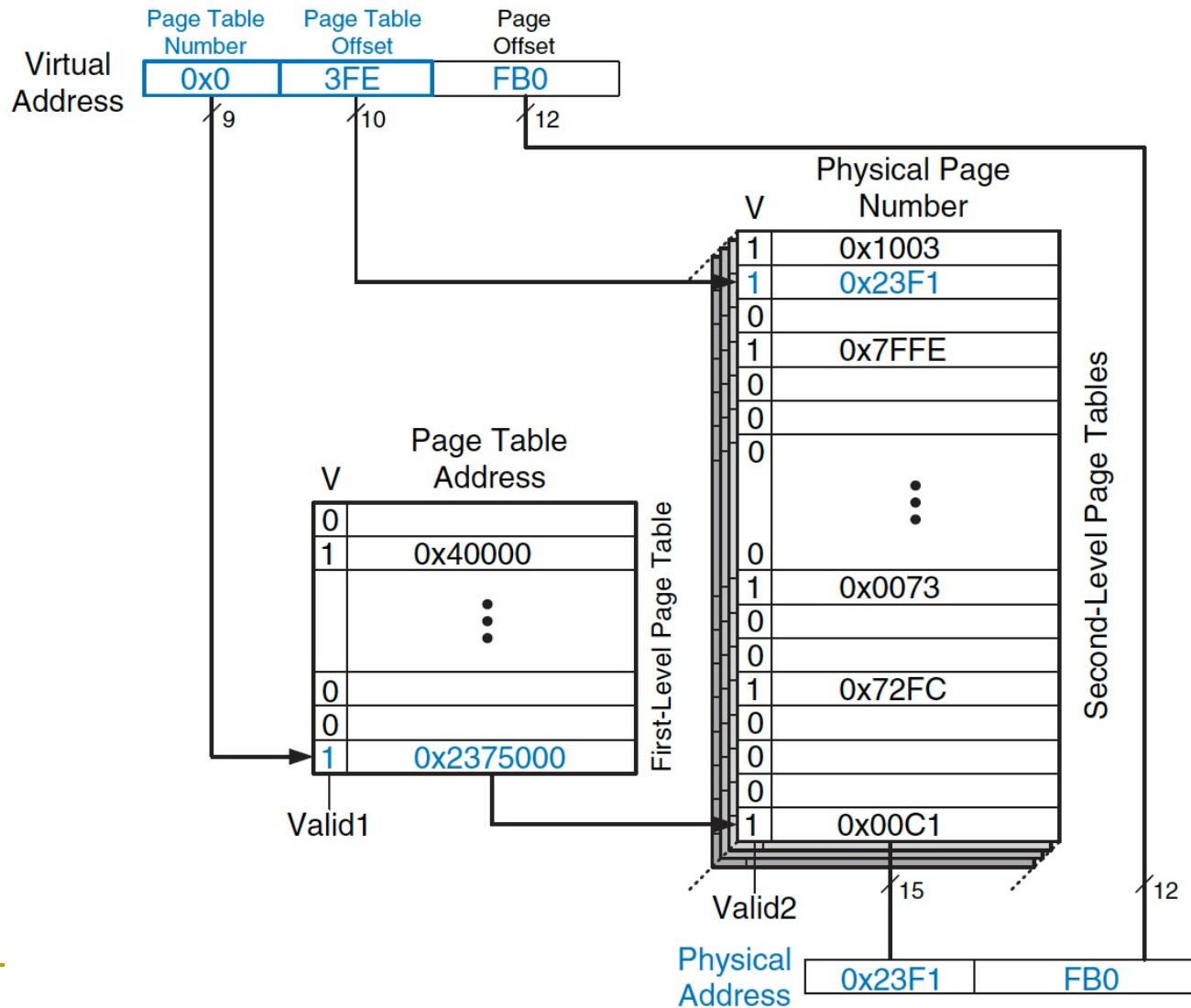
# Multi-Level Page Table Example

- First-level page table has to be in physical memory
- Only the needed second-level page tables can be kept in physical memory



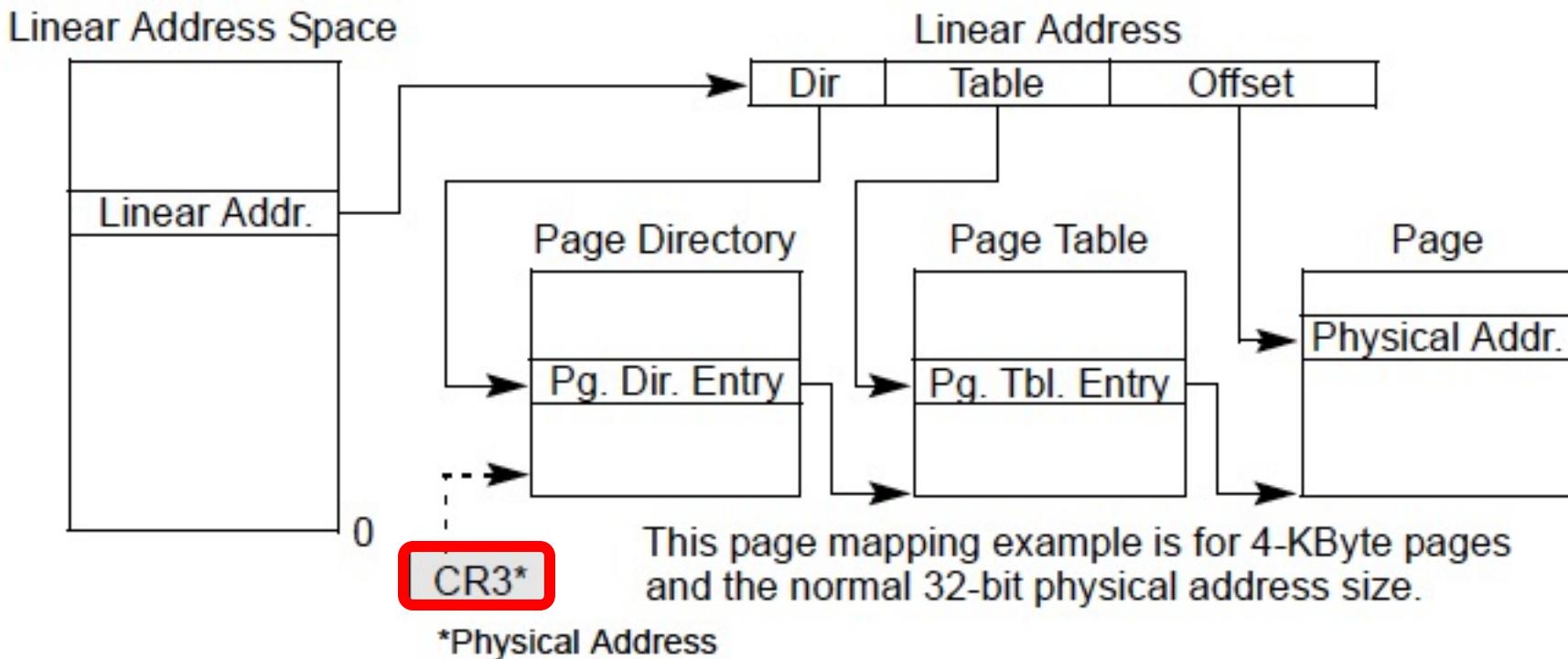
# Multi-Level Page Table: Address Translation

- For N-level page table, we need N page table accesses to find the PTE



# Multi-Level Page Tables from x86 Manual

Example from the x86 architecture



**CR3: Control Register 3 (or Page Directory Base Register)**

# x86 Page Tables (I): Small Pages

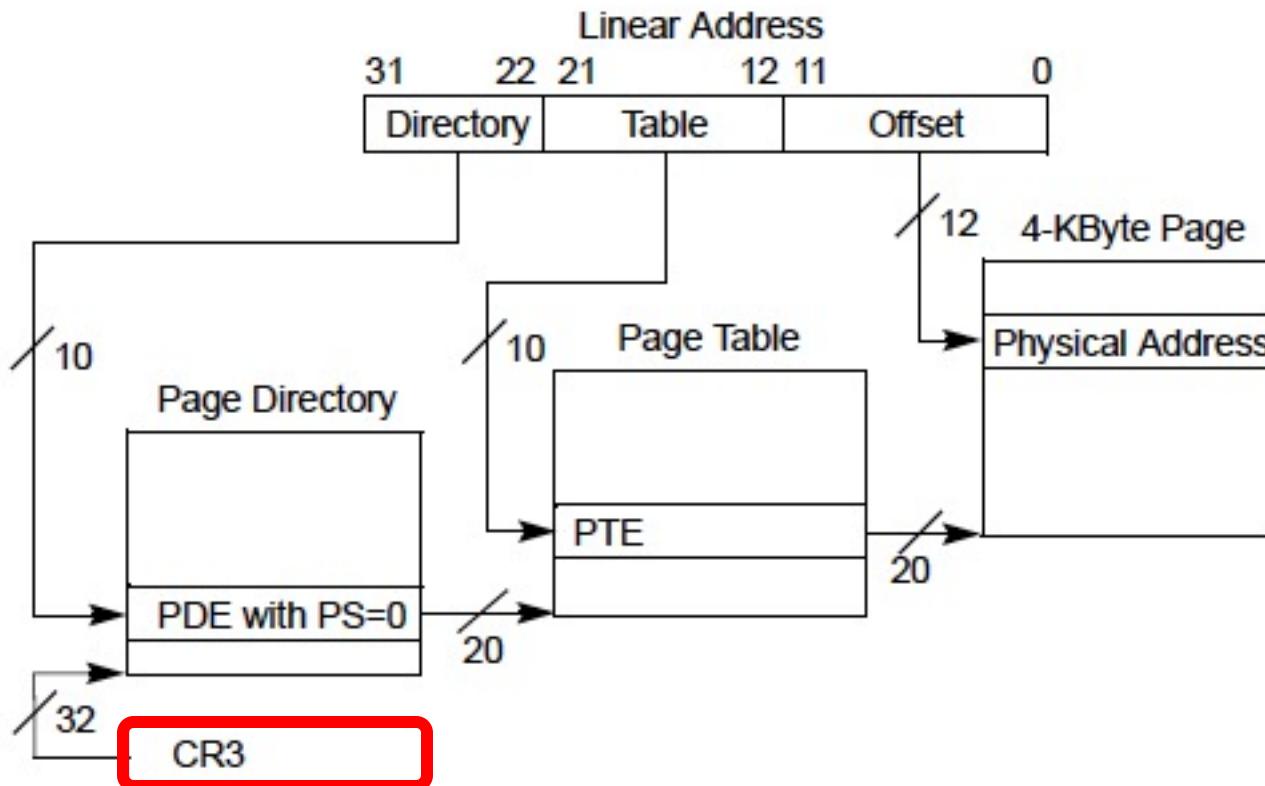


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

# x86 Page Tables (II): Large Pages

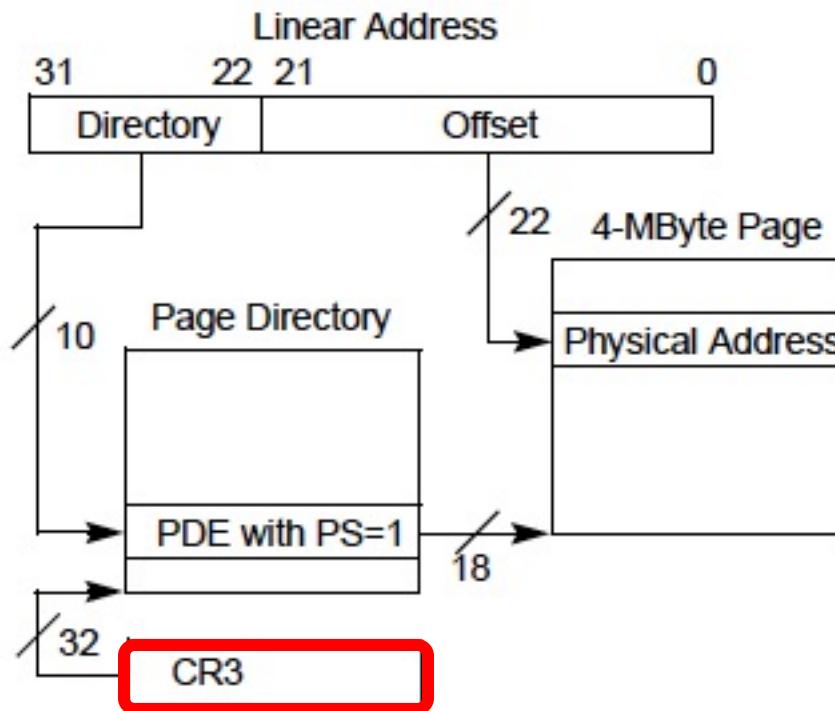


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

# Four-level Paging in x86-64

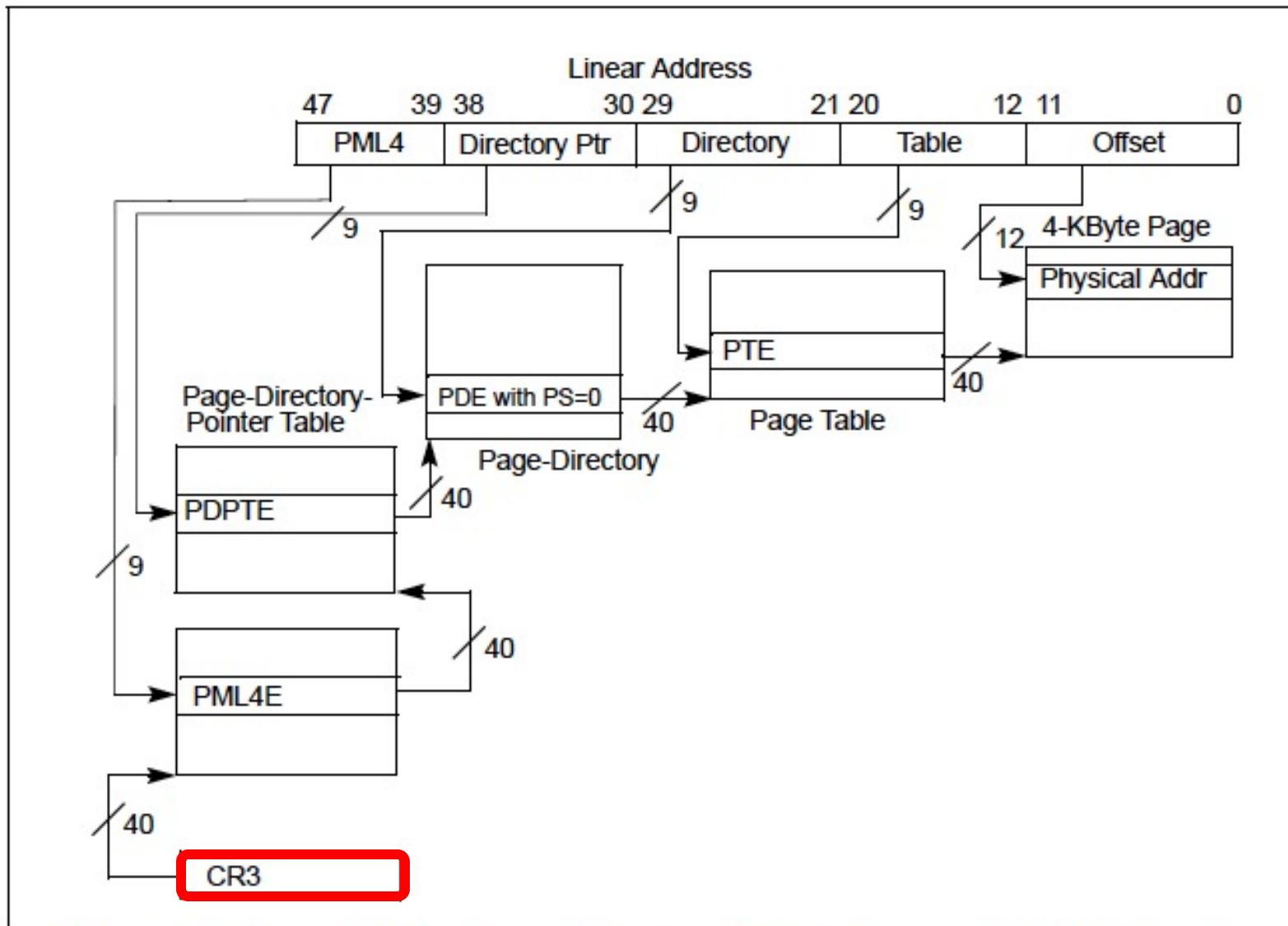


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

# Page Table Challenges (II)

---

- Challenge 1: **Page table is large**
  - at least part of it needs to be located in physical memory
  - solution: **multi-level (hierarchical) page tables**
- Challenge 2: **Each instruction fetch or load/store requires at least two memory accesses:**
  1. one for address translation (page table read)
  2. one to access data with the physical address (after translation)
- Two memory accesses to service an instruction fetch or load/store greatly degrades execution time
  - Num. of memory accesses increases with multi-level page tables
  - **Unless we are clever... → speed up the translation...**

# Translation Lookaside Buffer (TLB)

---

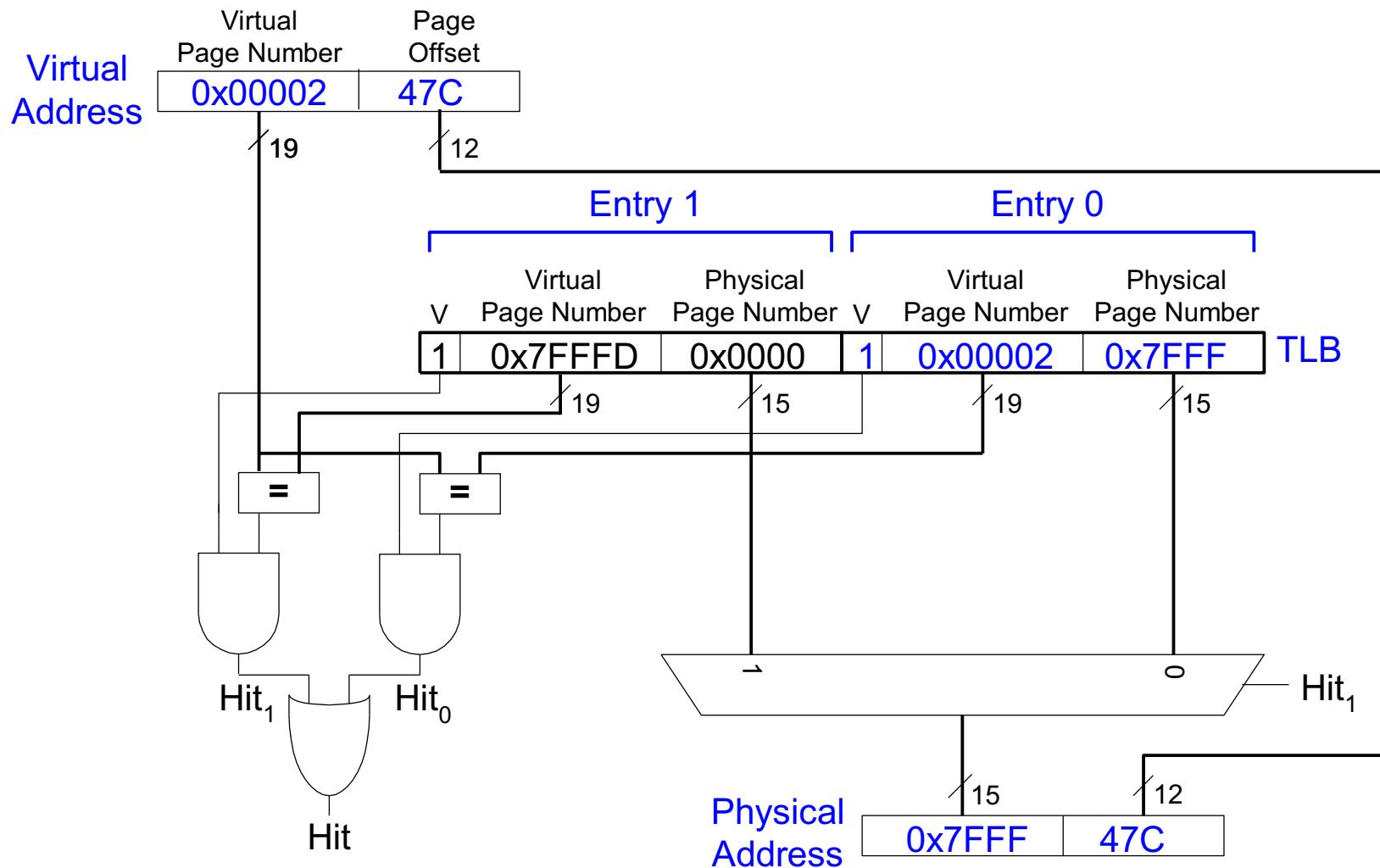
- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor to speed up address translation
- Translation lookaside buffer (TLB)
  - Small cache of most recently used translations (PTEs)
  - Reduces number of memory accesses required for *most* instruction fetches and loads/stores to only one

# Translation Lookaside Buffer (TLB)

---

- **Page table accesses have a lot of temporal locality**
  - Memory accesses have temporal and spatial locality
  - Large page sizes aid spatial locality (4KB, 8KB, MBs, GBs)
  - Consecutive instructions and loads/stores are likely to access same page
- **TLB: cache of page table entries (i.e., translations)**
  - Small: accessed in ~1 cycle
  - Typically 16 - 512 entries at level 1
  - Usually high associativity
  - > 90-99 % hit rates typical (depends on workload)
  - Reduces number of memory accesses for most instruction fetches and loads/stores to only one

# Example Two-Entry TLB



# TLB is a Translation (PTE) Cache

---

- All issues we discussed in caching and prefetching lectures apply to TLBs
- Example issues:
  - Instruction vs. Data TLBs
  - Multi-level TLBs
  - Associativity and size choices and tradeoffs
  - Insertion, promotion, replacement policies
  - What to keep in which TLB and how to decide that
  - Prefetching into the TLBs
  - TLB coherence
  - Shared vs. private TLBs across cores/threads
  - ...

# Virtual Memory Support and Examples

# Supporting Virtual Memory

---

- Virtual memory **requires both HW+SW support**
  - Page Table is in memory
  - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)
- The hardware component is called the **MMU** (memory management unit)
  - Includes Page Table Base Register(s), TLBs, page walkers
- **It is the job of the software** to leverage the MMU to
  - Populate page tables, decide what to replace in physical memory
  - Change the Page Table Base Register on context switch (to use the running thread's page table)
  - Handle page faults and ensure correct mapping

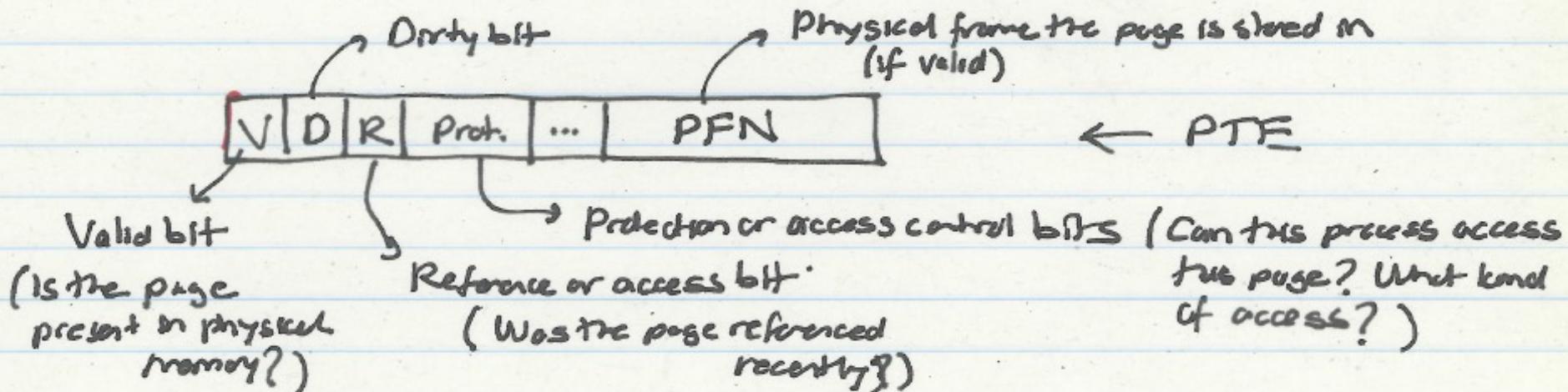
# Address Translation

---

- How to obtain the physical address from a virtual address?
- Page size specified by the ISA
  - VAX: 512 bytes
  - Today: 4KB, 8KB, 2GB, ... (small and large pages mixed together)
  - Trade-offs? (remember cache lectures)
- Page Table contains an entry for each virtual page
  - Called Page Table Entry (PTE)
  - What is in a PTE?

# What Is in a Page Table Entry (PTE)?

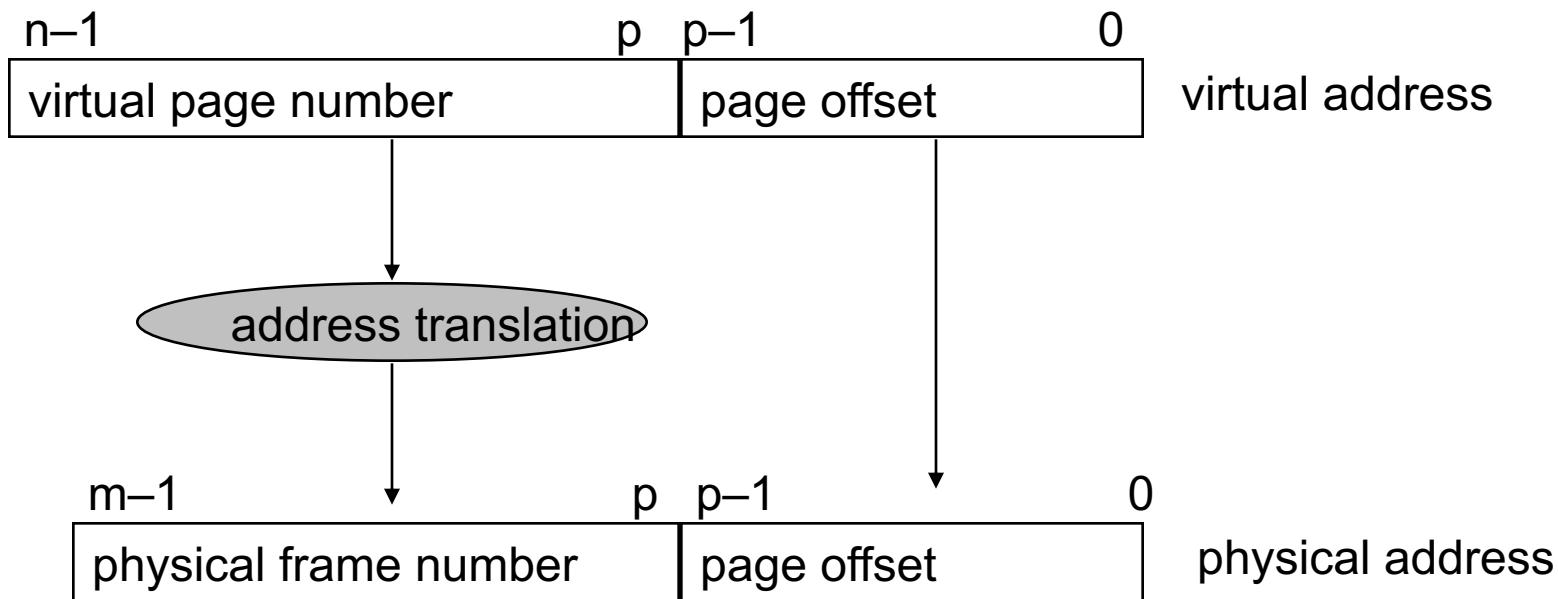
- Page table is the “tag store” for the physical memory data store
  - A mapping table between virtual memory and physical memory
- PTE is the “tag store entry” for a virtual page in memory
  - Need a **valid** bit → to indicate validity/presence in physical memory
  - Need **tag** bits (PFN) → to support translation
  - Need bits to support **replacement**
  - Need a **dirty** bit to support “write back caching”
  - Need **protection bits** to enable access control and protection



# Address Translation (I)

## ■ Parameters

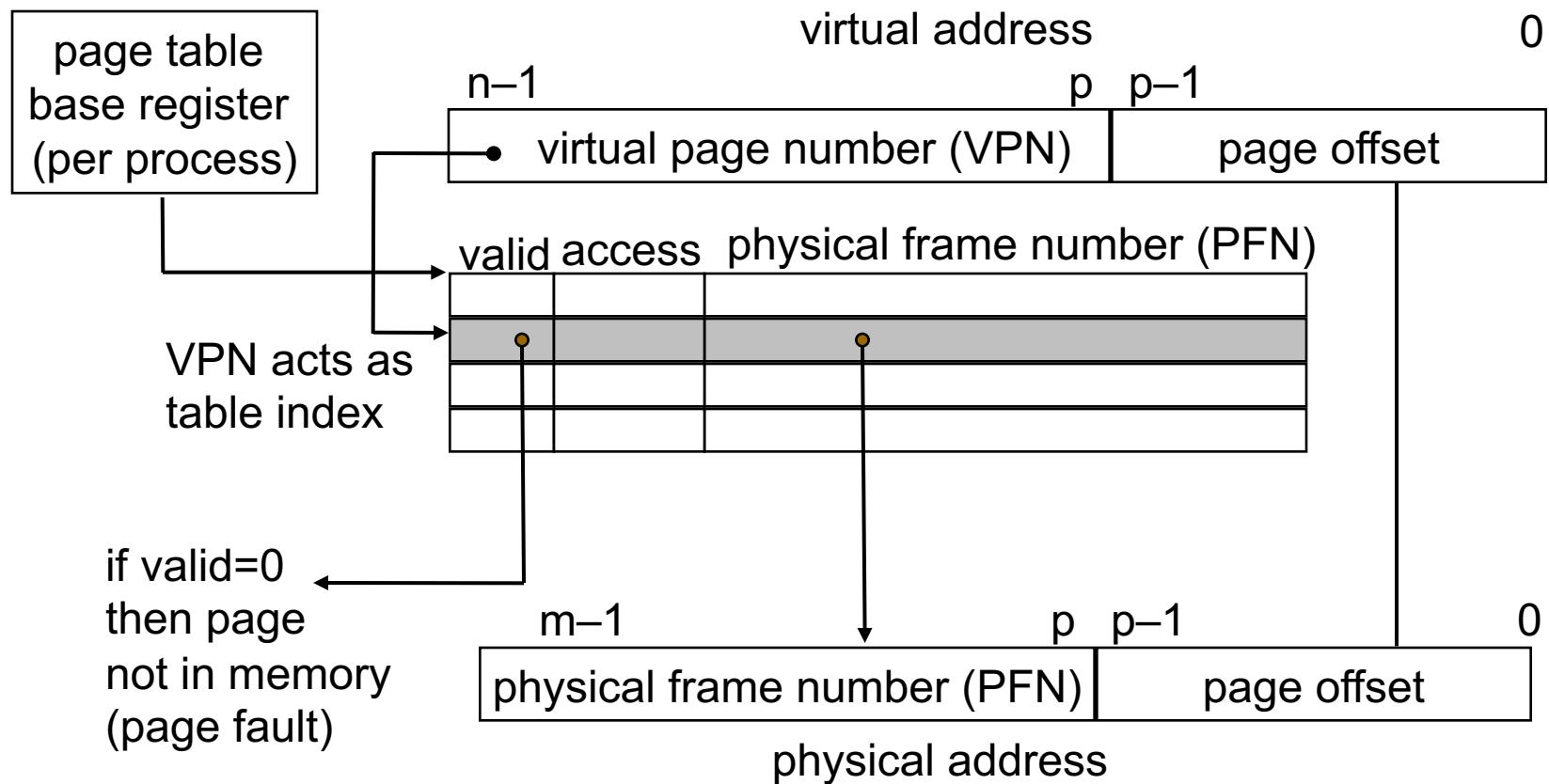
- ❑  $P = 2^p$  = page size (bytes).
- ❑  $N = 2^n$  = Virtual-address limit
- ❑  $M = 2^m$  = Physical-address limit



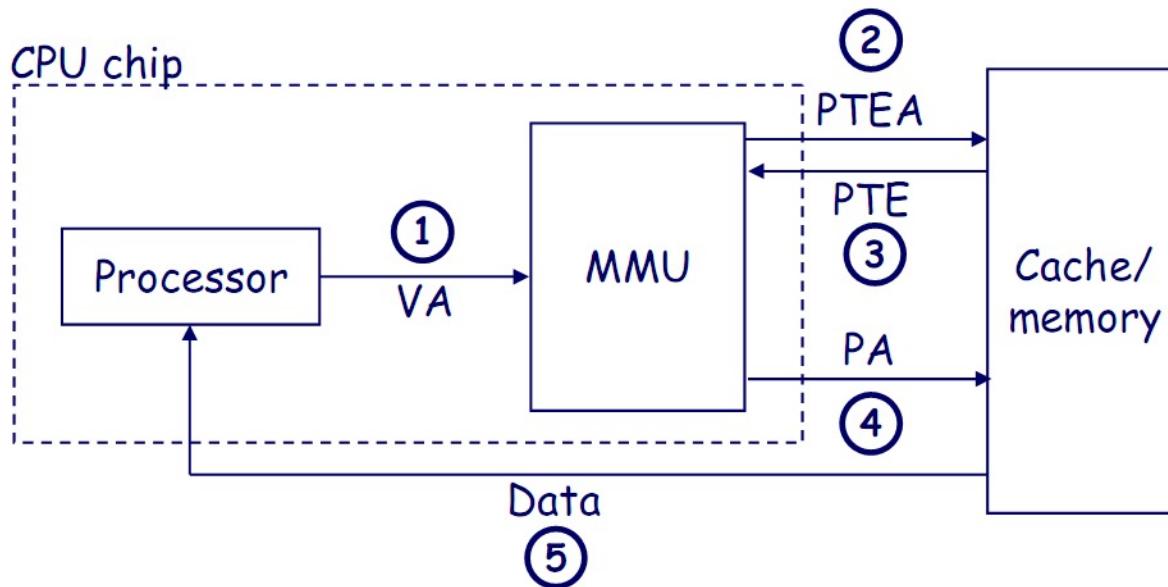
Page offset bits don't change as a result of translation

# Address Translation (II)

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page

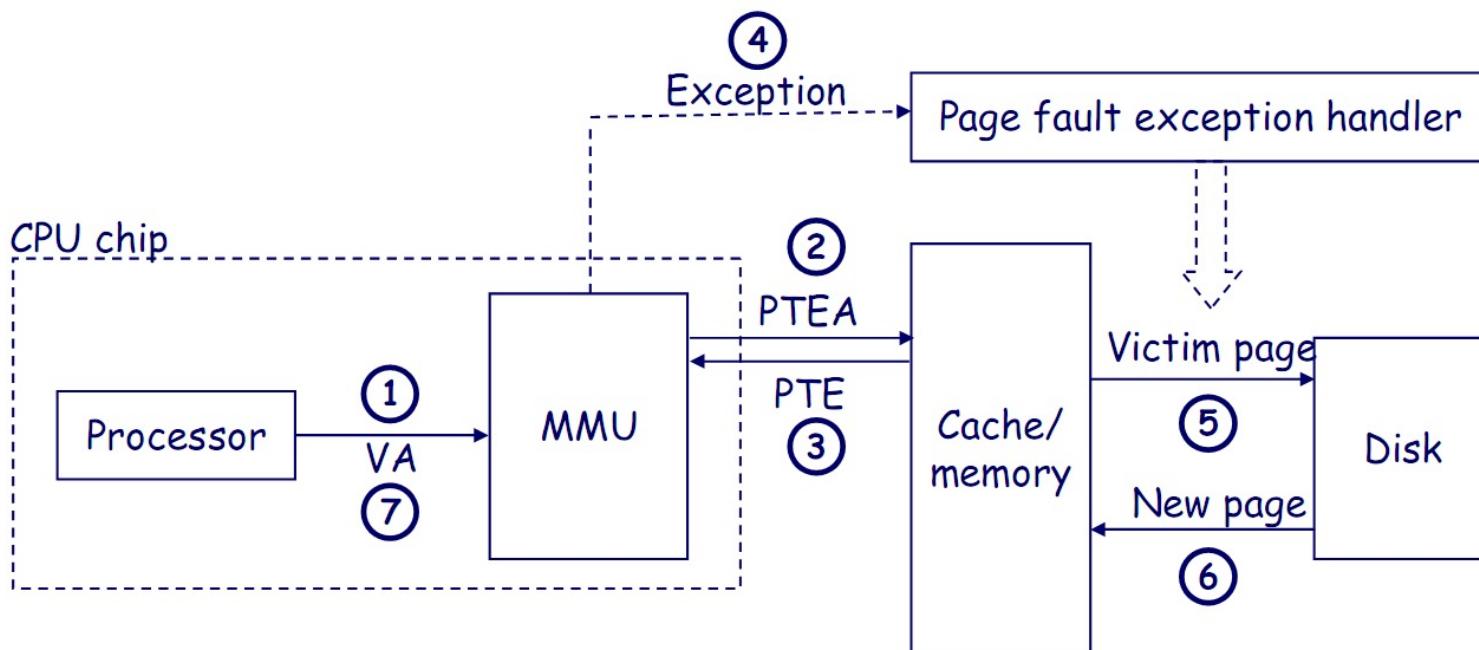


# Address Translation: Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to L1 cache
- 5) L1 cache sends data word to processor

# Address Translation: Page Fault

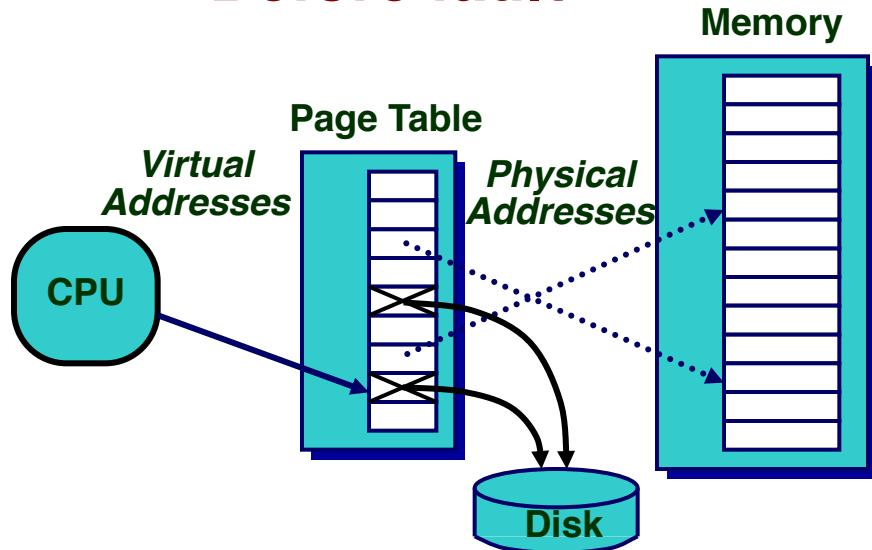


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim, and if dirty pages it out to disk
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction.

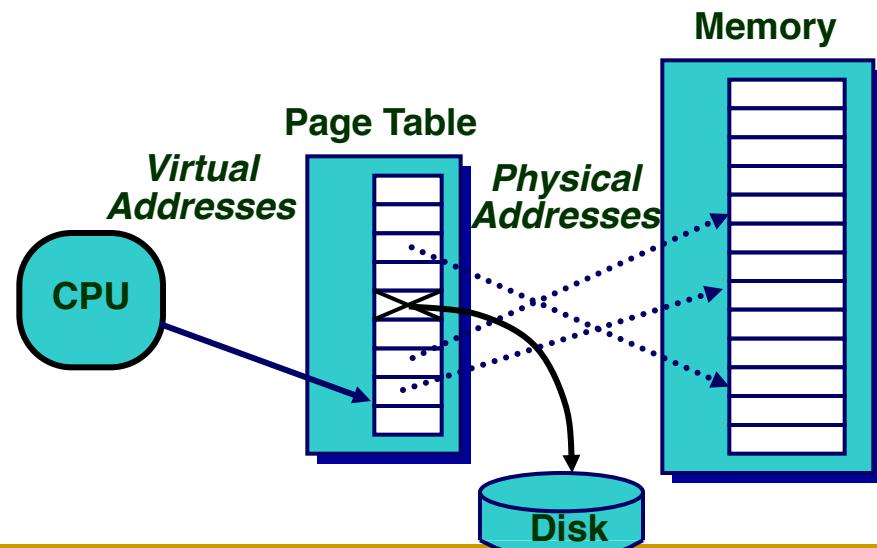
# Page Fault (“A Miss in Physical Memory”)

- If a page is not in physical memory but disk
  - Page table entry indicates virtual page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    - Other processes can continue executing
    - OS has full control over placement

**Before fault**

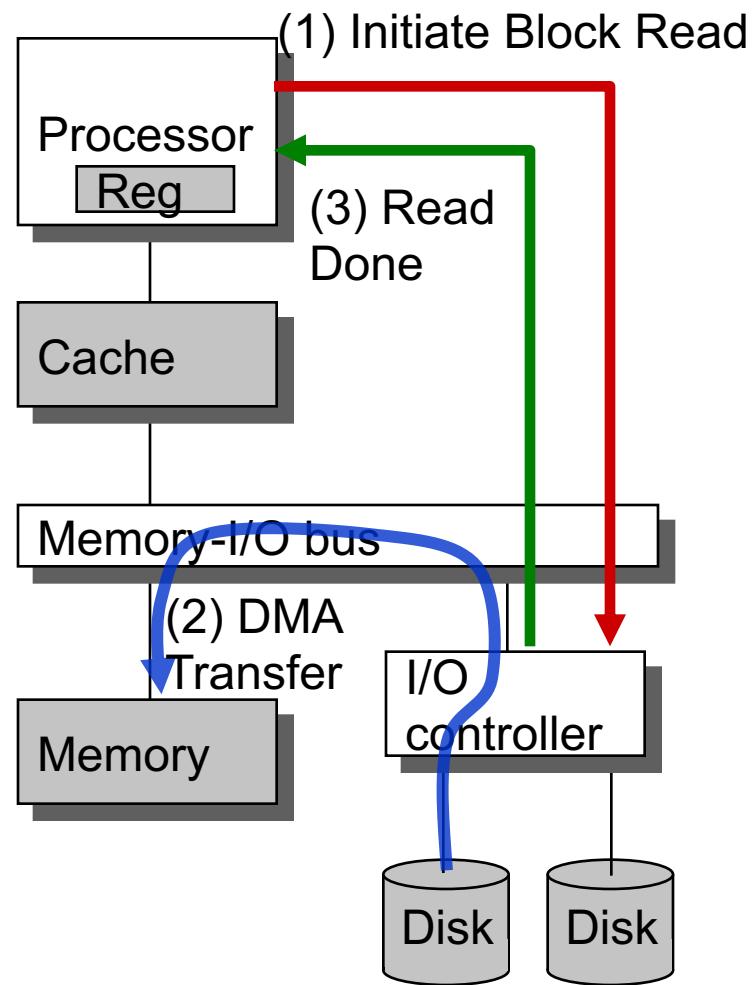


**After fault**



# Servicing a Page Fault

- (1) Processor signals controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- (2) Disk-to-mem read occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- (3) Controller signals completion
  - Interrupts processor
  - OS resumes suspended process



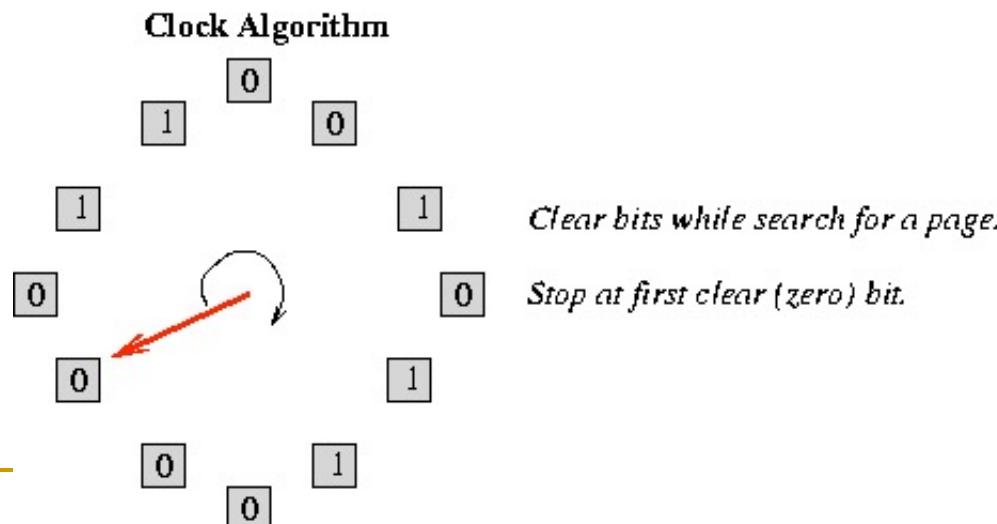
# Page Replacement Algorithms

---

- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?
- Is True LRU feasible?
  - 4GB memory, 4KB pages, how many possibilities of ordering?
- Modern systems use approximations of LRU
  - E.g., the CLOCK algorithm
- And, more sophisticated algorithms to take into account “frequency” of use
  - E.g., the ARC algorithm
  - Megiddo and Modha, “[ARC: A Self-Tuning, Low Overhead Replacement Cache](#),” FAST 2003.

# CLOCK Page Replacement Algorithm

- Keep a **circular list of physical frames** in memory (OS does)
- Keep a **pointer** (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
  - During traversal, clear the R bits of examined frames
  - Set the hand pointer to the next frame in the list



# Cache versus Page Replacement

---

- Physical memory (DRAM) is a cache for disk
  - Managed by system software via the virtual memory subsystem
- Page replacement is similar to cache replacement
- Page table is the “tag store” for physical memory data store
- What is the difference?
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - “Tolerable” amount of time to find a replacement candidate (disk versus memory access latency)
  - Role of hardware versus software

# Memory Protection

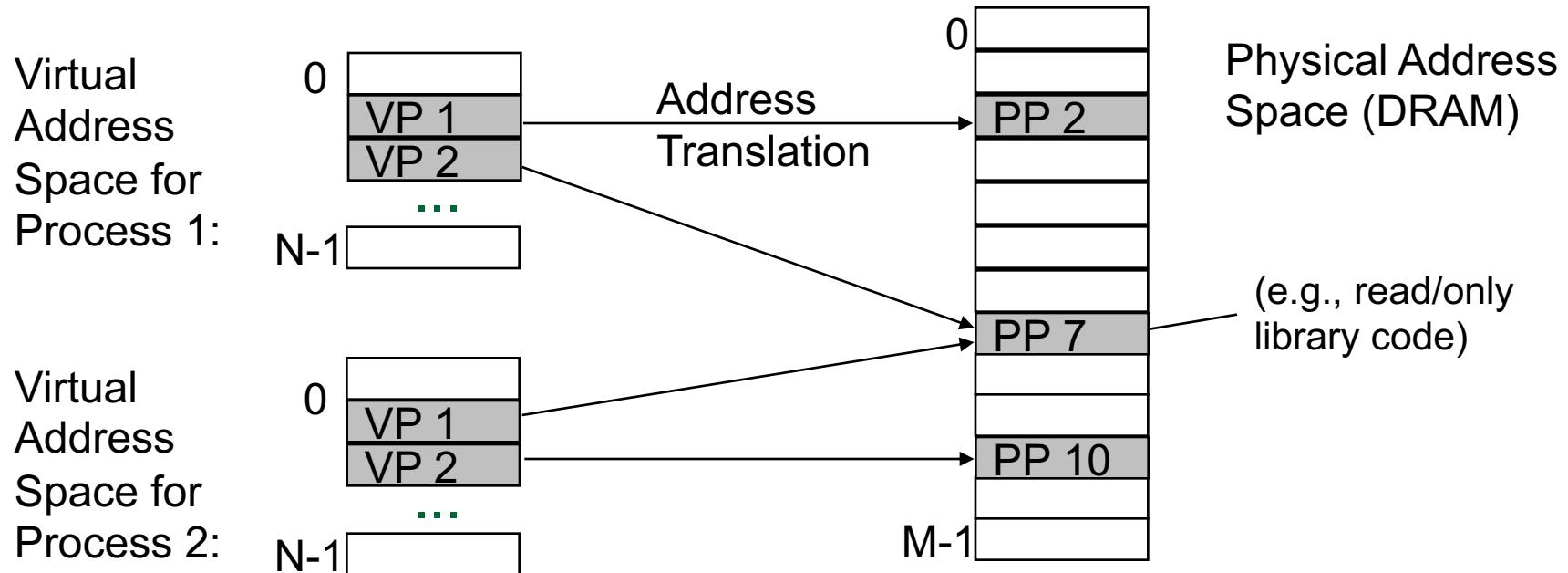
# Memory Protection

---

- Multiple programs (*processes*) run at once
  - Each process has its own page table
  - Each process can use entire virtual address space without worrying about where other programs are
  
- A process can only access physical pages mapped in its page table – cannot overwrite memory of another process
  - Provides protection and isolation between processes
  - Enables access control mechanisms per page

# Page Table is Per Process

- Each process has its own virtual address space
  - Full address space for each program
  - Simplifies memory allocation, sharing, linking and loading



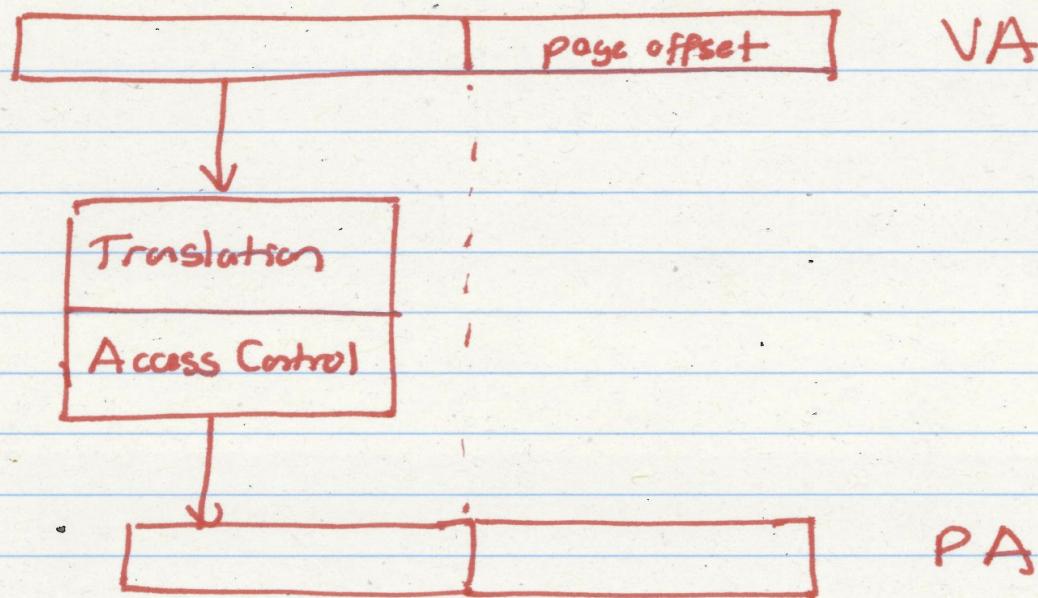
# Access Protection/Control via Virtual Memory

# Page-Level Access Control (Protection)

---

- Not every process is allowed to access every page
    - E.g., may need supervisor level privilege to access system pages
  - Idea: Store access control information on a page basis in the process's page table
  - Enforce access control at the same time as translation
- Virtual memory system serves two functions today
- Address translation (for illusion of large physical memory)
  - Access control (protection)

# Two Functions of Virtual Memory



Virtual  
Memory

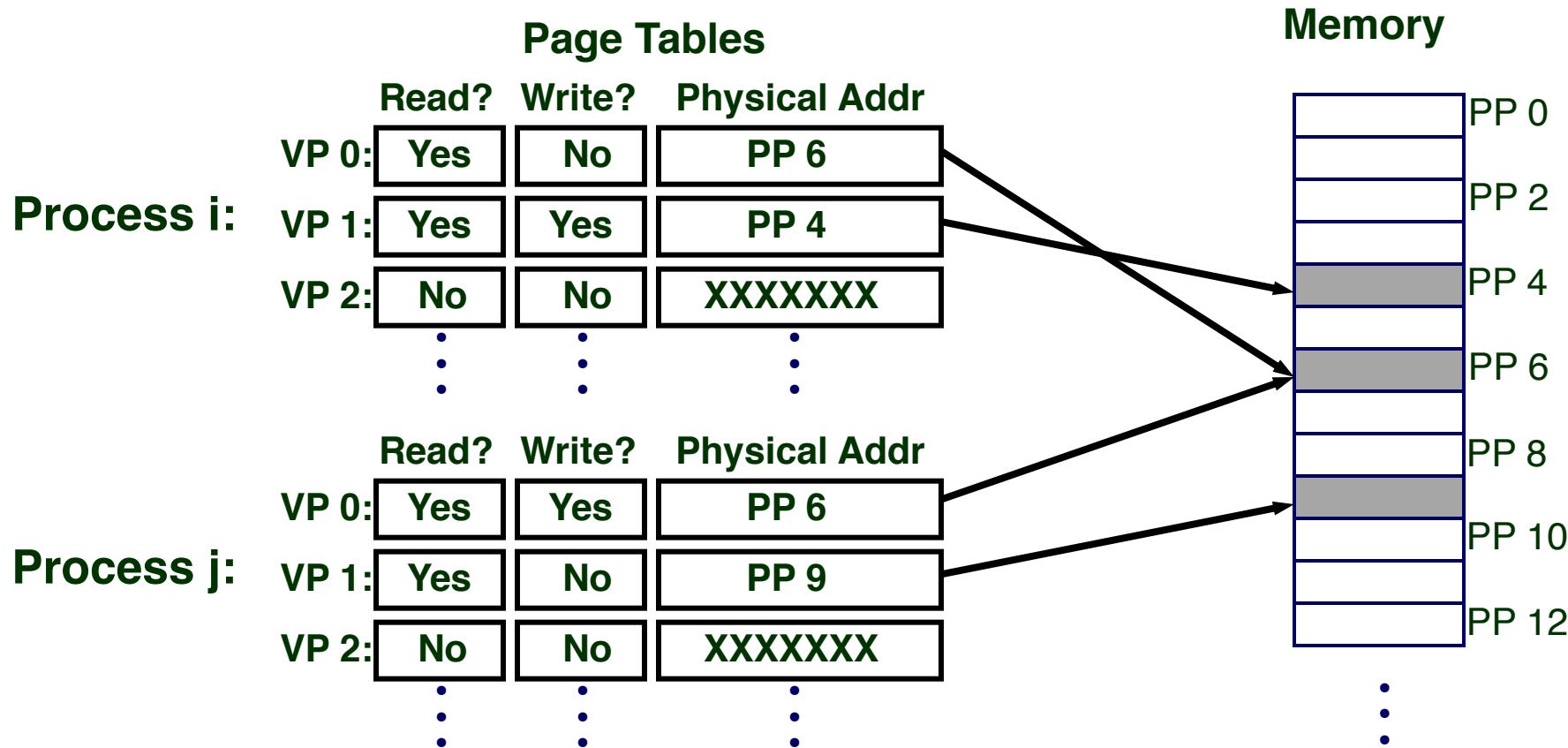
Two Functions  
Today

1. Translation
2. Access control  
(protection)

PTE contains access control bits associated with the virtual page.

# VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
  - If violated, generate exception (Access Protection exception)



# Privilege Levels in x86

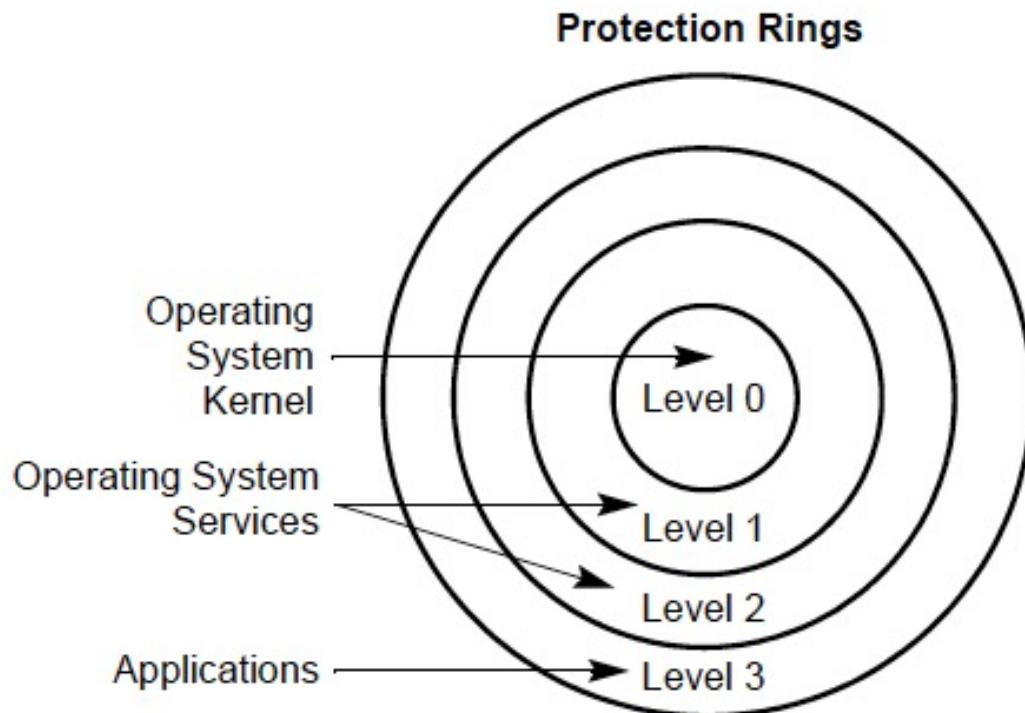


Figure 5-3. Protection Rings

# Privilege Levels in x86

---

- Four **privilege levels** in x86 (referred to as **rings**)

- Ring 0: Highest privilege (operating system)

**“Supervisor”**

- Ring 1: Not widely used

- Ring 2: Not widely used

- Ring 3: Lowest privilege (user applications)

**“User”**

# x86: A Closer Look at the PDE/PTE

- **PDE:** Page Directory Entry (32 bits)
- **PTE:** Page Table Entry (32 bits)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																																								
Address of page directory <sup>1</sup>												Ignored				P C D	Pw T	Ignored				CR3																		
Bits 31:22 of address of 2MB page frame				Reserved (must be 0)				Bits 39:32 of address <sup>2</sup>				P A T	Ignored	G	1	D	A	P C D	Pw T	U S	R W	1	PDE: 4MB page																	
PDE & PTE Address of page table																0	I g n o r e	1 g n a d r	P P W L	U U R	1	PDE: page table																		
Ignored																0	PDE: not present																							
PTE Address of 4KB page frame																Ignored	G P A T	P P W L	U U R /W	1	PTE: 4KB page																			
Ignored																0	PTE: not present																							

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

# Protection: PDE's Flags

- Protects all 1024 pages in a page table

Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored

# Protection: PTE's Flags

## ■ Protects one page at a time

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0) <sup>1</sup>
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise

# Page Level Protection in x86

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

# Protection: PDE + PTE = ???

Table 5-3. Combined Page-Directory and Page-Table Protection

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only		Read-Write	User	Read-Only
User	Read-Write		Read-Only	User	Read-Only
User	Read-Write		Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

**NOTE:**

- \* If CR0.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CR0.WP = 0, supervisor privilege permits read-write access.

# Food for Thought: What If?

---

- Your hardware is unreliable and someone can flip the access protection bits
  - such that a user-level program can gain supervisor-level access (i.e., access to all data on the system)
  - by flipping the access control bit from user to supervisor!
- Can this happen?

# Remember RowHammer?

---

One can  
predictably induce errors  
in most DRAM memory chips

# Remember RowHammer?

- One can predictably induce bit flips in commodity DRAM chips
  - >80% of the tested DRAM chips are vulnerable
- First example of how a simple hardware failure mechanism can create a widespread system security vulnerability

WIRED

Forget Software—Now Hackers Are Exploiting Physics

BUSINESS

CULTURE

DESIGN

GEAR

SCIENCE

ANDY GREENBERG SECURITY 08.31.16 7:00 AM

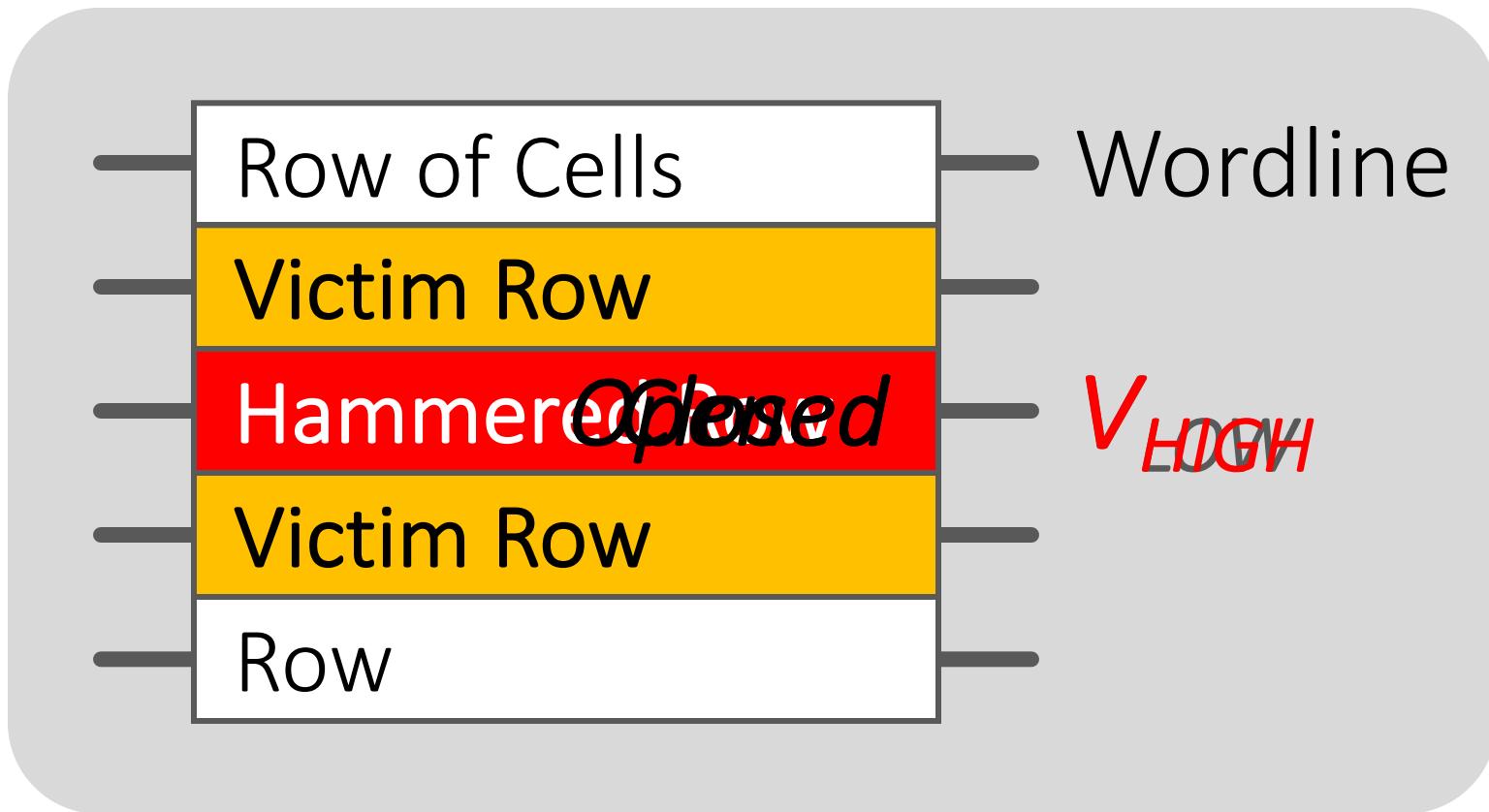
SHARE

 SHARE  
18276

 TWEET

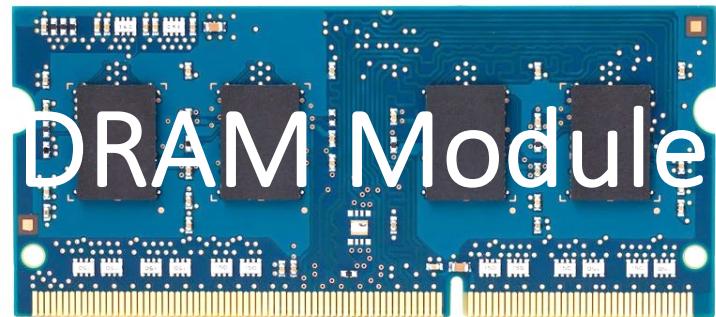
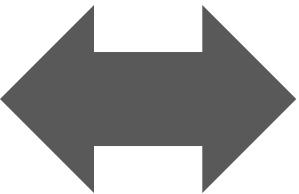
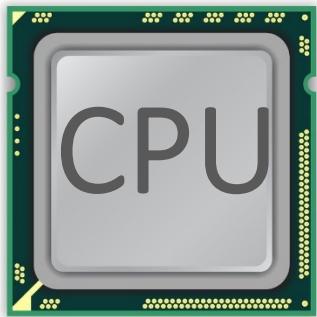
# FORGET SOFTWARE—NOW HACKERS ARE EXPLOITING PHYSICS

# Modern DRAM is Prone to Disturbance Errors

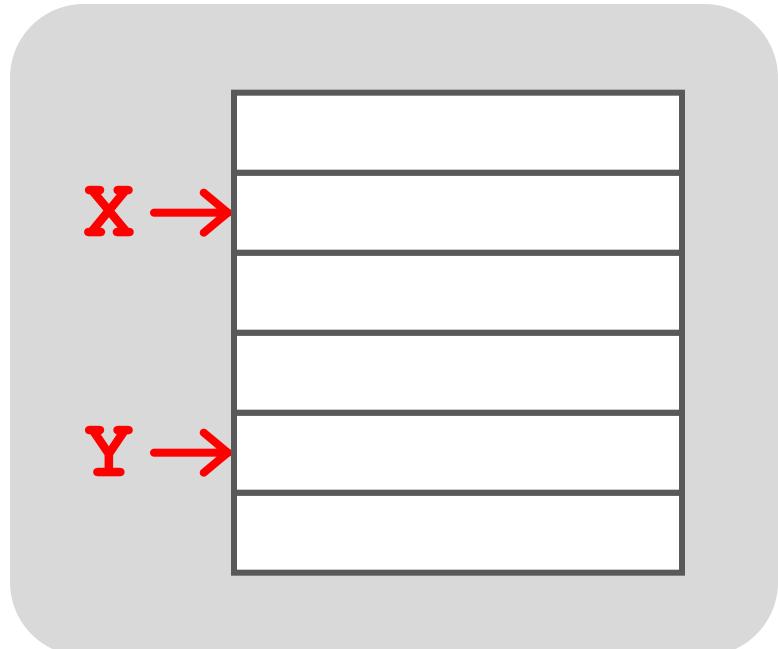


Repeatedly reading a row enough times (before memory gets refreshed) induces **disturbance errors** in adjacent rows in **most real DRAM chips you can buy today**

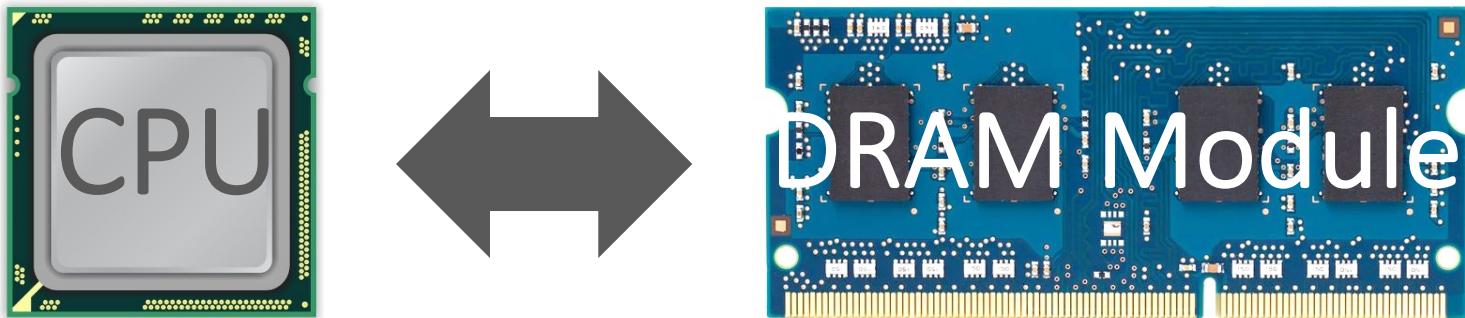
# A Simple Program Can Induce Many Errors



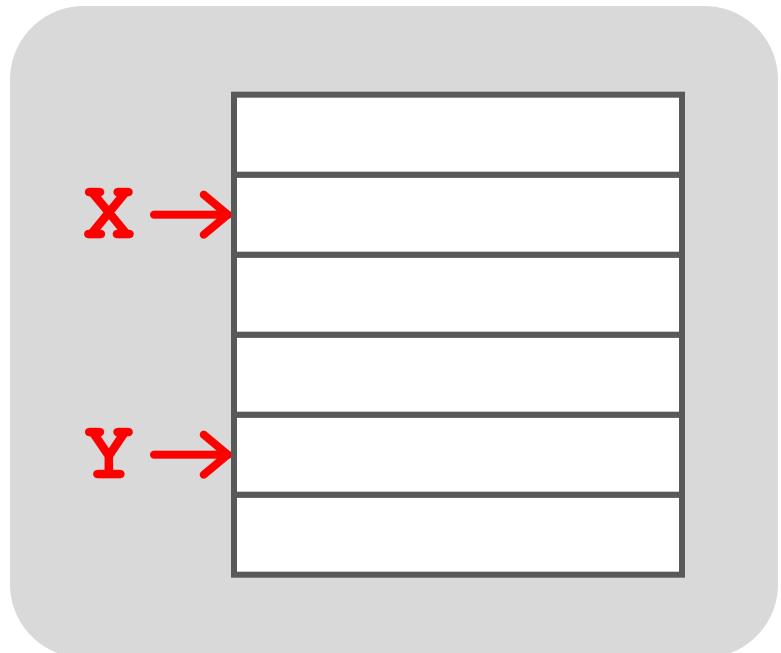
```
loop:  
    mov (%X), %eax  
    mov (%Y), %ebx  
    clflush (%X)  
    clflush (%Y)  
    mfence  
    jmp loop
```



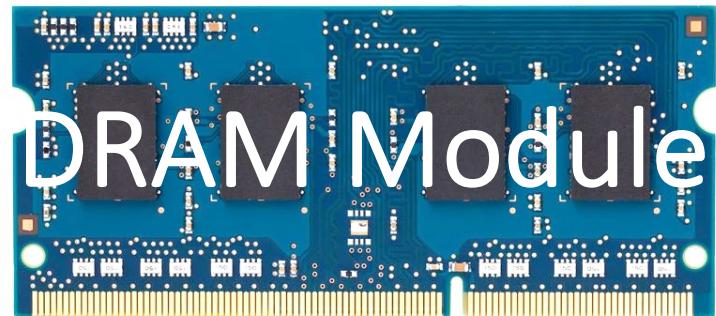
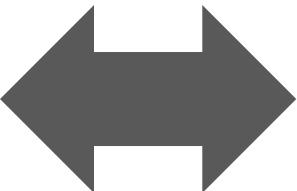
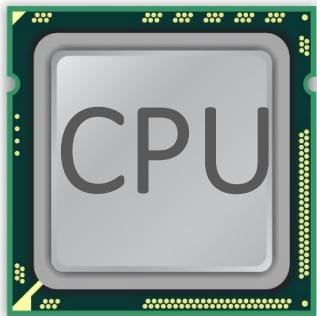
# A Simple Program Can Induce Many Errors



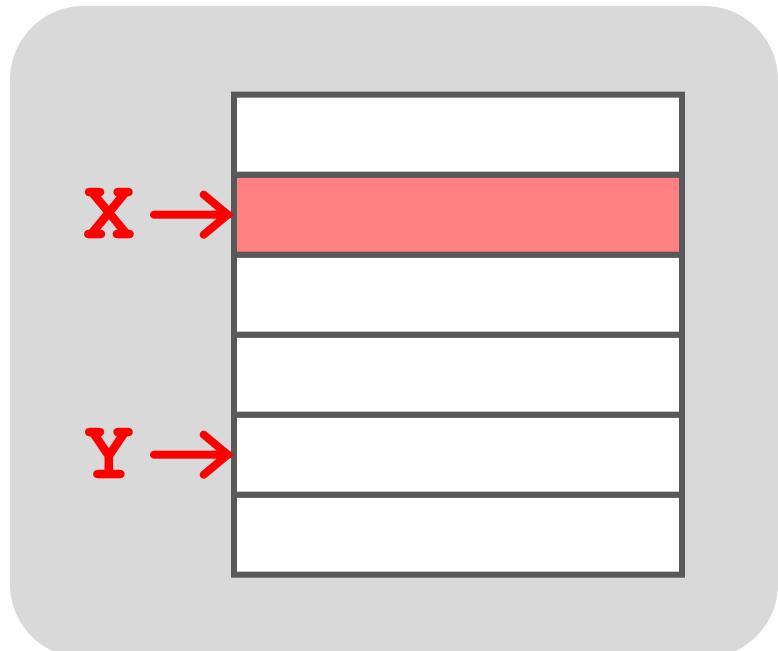
1. Avoid *cache hits*
  - Flush **X** from cache
2. Avoid *row hits* to **X**
  - Read **Y** in another row



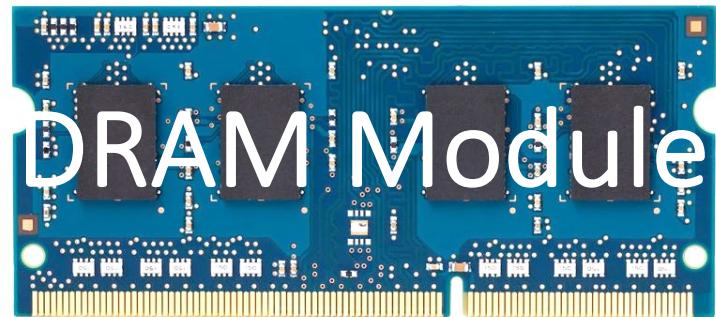
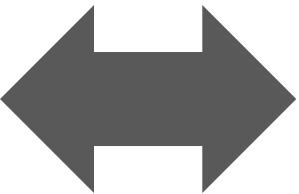
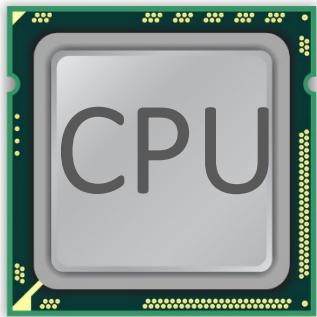
# A Simple Program Can Induce Many Errors



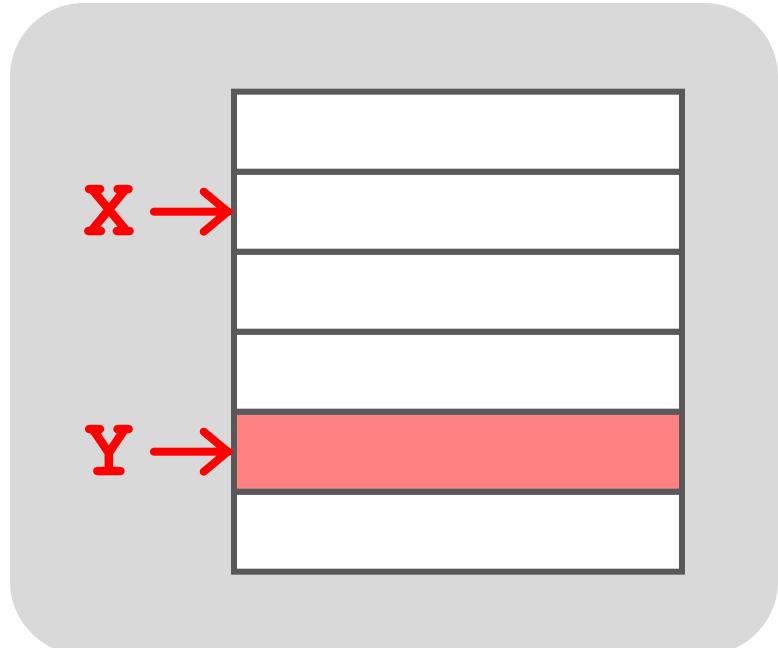
```
loop:  
    mov (%X), %eax  
    mov (%Y), %ebx  
    clflush (%X)  
    clflush (%Y)  
    mfence  
    jmp loop
```



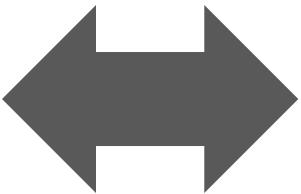
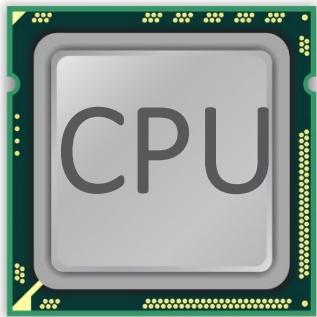
# A Simple Program Can Induce Many Errors



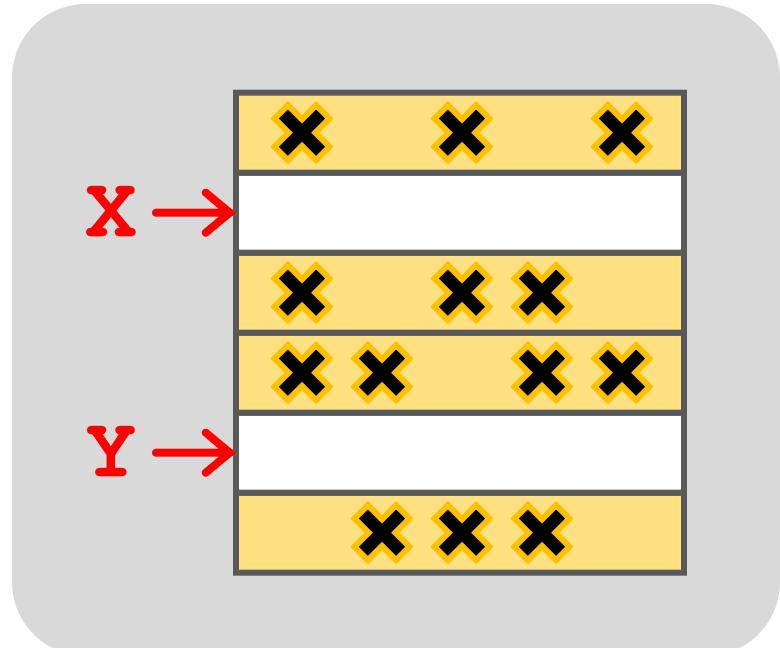
```
loop:  
    mov (%X), %eax  
    mov (%Y), %ebx  
    clflush (%X)  
    clflush (%Y)  
    mfence  
    jmp loop
```



# A Simple Program Can Induce Many Errors



```
loop:  
    mov (%X), %eax  
    mov (%Y), %ebx  
    clflush (%X)  
    clflush (%Y)  
    mfence  
    jmp loop
```



# Observed Errors in Real Systems

CPU Architecture	Errors	Access-Rate
Intel Haswell (2013)	22.9K	12.3M/sec
Intel Ivy Bridge (2012)	20.7K	11.7M/sec
Intel Sandy Bridge (2011)	16.1K	11.6M/sec
AMD Piledriver (2012)	59	6.1M/sec

A real reliability & security issue

# One Can Take Over an Otherwise-Secure System

---

## Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

*Abstract. Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology*

# Project Zero

[Flipping Bits in Memory Without Accessing Them:  
An Experimental Study of DRAM Disturbance Errors](#)  
(Kim et al., ISCA 2014)

News and updates from the Project Zero team at Google

[Exploiting the DRAM rowhammer bug to  
gain kernel privileges](#) (Seaborn, 2015)

Monday, March 9, 2015

Exploiting the DRAM rowhammer bug to gain kernel privileges

# RowHammer Security Attack Example

---

- “Rowhammer” is a problem with some recent DRAM devices in which repeatedly accessing a row of memory can cause bit flips in adjacent rows (Kim et al., ISCA 2014).
  - Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors (Kim et al., ISCA 2014)
- We tested a selection of laptops and found that a subset of them exhibited the problem.
- We built two working privilege escalation exploits that use this effect.
  - Exploiting the DRAM rowhammer bug to gain kernel privileges (Seaborn+, 2015)
- One exploit uses rowhammer-induced bit flips to gain kernel privileges on x86-64 Linux when run as an unprivileged userland process.
- When run on a machine vulnerable to the rowhammer problem, the process was able to induce bit flips in page table entries (PTEs).
- It was able to use this to gain write access to its own page table, and hence gain read-write access to all of physical memory.

# Security Implications

Rowhammer



# Security Implications



It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after

# More Security Implications (I)

**"We can gain unrestricted access to systems of website visitors."**

www.iaik.tugraz.at ■

Not there yet, but ...



ROOT privileges for web apps!

29

Daniel Gruss (@lavados), Clémentine Maurice (@BloodyTangerine),  
December 28, 2015 — 32c3, Hamburg, Germany



Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript (DIMVA'16)

# More Security Implications (II)

**“Can gain control of a smart phone deterministically”**



Drammer: Deterministic Rowhammer  
Attacks on Mobile Platforms, CCS'16 91

# More Security Implications (III)

- Using an integrated GPU in a mobile system to remotely escalate privilege via the WebGL interface

The image shows a snippet from Ars Technica's website. At the top, there's a navigation bar with the site's logo and categories: BIZ & IT, TECH, SCIENCE, POLICY, CARS, and GAMING & CULTURE. Below the navigation bar, the headline reads: "GRAND PWNING UNIT" — Drive-by Rowhammer attack uses GPU to compromise an Android phone. A subtitle below the headline states: "JavaScript based GLitch pwns browsers by flipping bits inside memory chips." At the bottom of the snippet, it says "DAN GOODIN - 5/3/2018, 12:00 PM".

"GRAND PWNING UNIT" —

## Drive-by Rowhammer attack uses GPU to compromise an Android phone

JavaScript based GLitch pwns browsers by flipping bits inside memory chips.

DAN GOODIN - 5/3/2018, 12:00 PM

## Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU

Pietro Frigo  
Vrije Universiteit  
Amsterdam  
p.frigo@vu.nl

Cristiano Giuffrida  
Vrije Universiteit  
Amsterdam  
giuffrida@cs.vu.nl

Herbert Bos  
Vrije Universiteit  
Amsterdam  
herbertb@cs.vu.nl

Kaveh Razavi  
Vrije Universiteit  
Amsterdam  
kaveh@cs.vu.nl

# More Security Implications (IV)

- Rowhammer over RDMA (I)

**ars** TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

THROWHAMMER —

## Packets over a LAN are all it takes to trigger serious Rowhammer bit flips

The bar for exploiting potentially serious DDR weakness keeps getting lower.

DAN GOODIN - 5/10/2018, 5:26 PM

### Throwhammer: Rowhammer Attacks over the Network and Defenses

Andrei Tatar  
*VU Amsterdam*

Radhesh Krishnan  
*VU Amsterdam*

Elias Athanasopoulos  
*University of Cyprus*

Cristiano Giuffrida  
*VU Amsterdam*

Herbert Bos  
*VU Amsterdam*

Kaveh Razavi  
*VU Amsterdam*

# More Security Implications (V)

## ■ Rowhammer over RDMA (II)



Nethammer—Exploiting DRAM Rowhammer Bug Through Network Requests



## **Nethammer: Inducing Rowhammer Faults through Network Requests**

Moritz Lipp  
Graz University of Technology

Daniel Gruss  
Graz University of Technology

Misiker Tadesse Aga  
University of Michigan

Clémentine Maurice  
Univ Rennes, CNRS, IRISA

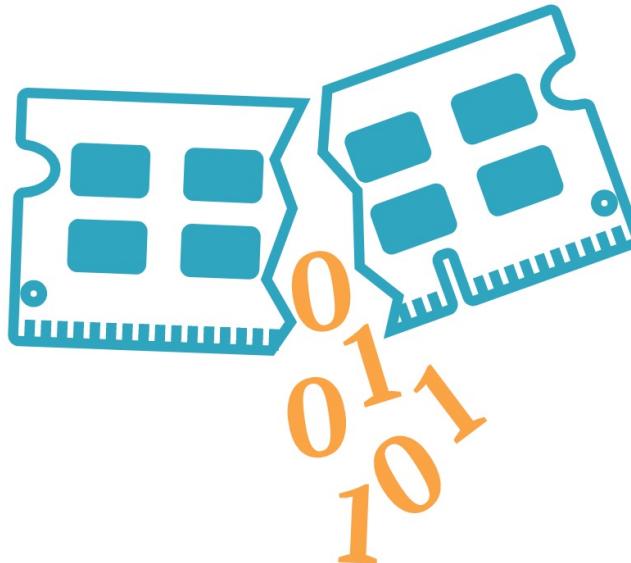
Michael Schwarz  
Graz University of Technology

Lukas Raab  
Graz University of Technology

Lukas Lamster  
Graz University of Technology

# More Security Implications (VI)

- IEEE S&P 2020



## RAMBleed

### RAMBleed: Reading Bits in Memory Without Accessing Them

Andrew Kwong

*University of Michigan*

[ankwong@umich.edu](mailto:ankwong@umich.edu)

Daniel Genkin

*University of Michigan*

[genkin@umich.edu](mailto:genkin@umich.edu)

Daniel Gruss

*Graz University of Technology*

[daniel.gruss@iaik.tugraz.at](mailto:daniel.gruss@iaik.tugraz.at)

Yuval Yarom

*University of Adelaide and Data61*

[yval@cs.adelaide.edu.au](mailto:yval@cs.adelaide.edu.au)

# More Security Implications (VII)

- USENIX Security 2019

## Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks

Sanghyun Hong, Pietro Frigo<sup>†</sup>, Yiğitcan Kaya, Cristiano Giuffrida<sup>†</sup>, Tudor Dumitraş

*University of Maryland, College Park*

*†Vrije Universiteit Amsterdam*



### A Single Bit-flip Can Cause Terminal Brain Damage to DNNs

*One specific bit-flip in a DNN's representation leads to accuracy drop over 90%*

Our research found that a specific bit-flip in a DNN's bitwise representation can cause the accuracy loss up to 90%, and the DNN has 40-50% parameters, on average, that can lead to the accuracy drop over 10% when individually subjected to such single bitwise corruptions...

[Read More](#)

# More Security Implications (VIII)

## ■ USENIX Security 2020

### DeepHammer: Depleting the Intelligence of Deep Neural Networks through Targeted Chain of Bit Flips

Fan Yao

*University of Central Florida*  
*fan.yao@ucf.edu*

Adnan Siraj Rakin

*Arizona State University*  
*asrakin@asu.edu*

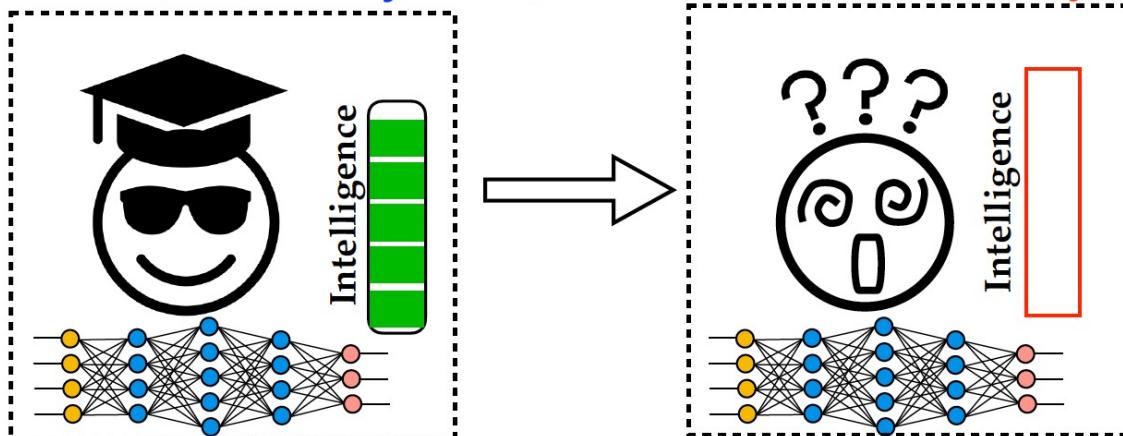
Deliang Fan

*dfan@asu.edu*

Degrade the inference accuracy to the level of Random Guess

Example: ResNet-20 for CIFAR-10, 10 output classes

Before attack, **Accuracy: 90.2%** After attack, **Accuracy: ~10% (1/10)**



# More Security Implications?

---



# Curious? First RowHammer Paper

---

- Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu,

## **"Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors"**

*Proceedings of the 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, June 2014.*

[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Source Code and Data]

## **Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors**

Yoongu Kim<sup>1</sup>   Ross Daly\*   Jeremie Kim<sup>1</sup>   Chris Fallin\*   Ji Hye Lee<sup>1</sup>  
Donghyuk Lee<sup>1</sup>   Chris Wilkerson<sup>2</sup>   Konrad Lai   Onur Mutlu<sup>1</sup>

<sup>1</sup>Carnegie Mellon University

<sup>2</sup>Intel Labs

# Curious? RowHammer: Now and Beyond...

---

- Onur Mutlu and Jeremie Kim,

**["RowHammer: A Retrospective"](#)**

*IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) Special Issue on Top Picks in Hardware and Embedded Security, 2019.*

[[Preliminary arXiv version](#)]

[[Slides from COSADE 2019 \(pptx\)](#)]

[[Slides from VLSI-SOC 2020 \(pptx\) \(pdf\)](#)]

[[Talk Video \(30 minutes\)](#)]

## RowHammer: A Retrospective

Onur Mutlu<sup>§‡</sup>

<sup>§</sup>ETH Zürich

Jeremie S. Kim<sup>†§</sup>

<sup>†</sup>Carnegie Mellon University

# RowHammer in 2020 (I)

---

- Jeremie S. Kim, Minesh Patel, A. Giray Yaglikci, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu,  
**"Revisiting RowHammer: An Experimental Analysis of Modern Devices and Mitigation Techniques"**

*Proceedings of the 47th International Symposium on Computer Architecture (ISCA), Valencia, Spain, June 2020.*

[[Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (20 minutes)]

[[Lightning Talk Video](#) (3 minutes)]

## Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques

Jeremie S. Kim<sup>§†</sup>      Minesh Patel<sup>§</sup>      A. Giray Yağlıkçı<sup>§</sup>  
Hasan Hassan<sup>§</sup>      Roknoddin Azizi<sup>§</sup>      Lois Orosa<sup>§</sup>      Onur Mutlu<sup>§†</sup>

<sup>§</sup>*ETH Zürich*

<sup>†</sup>*Carnegie Mellon University*

# RowHammer in 2020 (II)

---

- Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi,

## "**TRRespass: Exploiting the Many Sides of Target Row Refresh**"

*Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 2020.*

[[Slides \(pptx\)](#) ([pdf](#))]

[[Lecture Slides \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (17 minutes)]

[[Lecture Video](#) (59 minutes)]

[[Source Code](#)]

[[Web Article](#)]

**Best paper award.**

**Pwnie Award 2020 for Most Innovative Research.** [Pwnie Awards 2020](#)

# TRRespass: Exploiting the Many Sides of Target Row Refresh

Pietro Frigo\*†    Emanuele Vannacci\*†    Hasan Hassan§    Victor van der Veen¶  
Onur Mutlu§    Cristiano Giuffrida\*    Herbert Bos\*    Kaveh Razavi\*

# RowHammer in 2020 (III)

---

- Lucian Cojocar, Jeremie Kim, Minesh Patel, Lillian Tsai, Stefan Saroiu, Alec Wolman, and Onur Mutlu,

## **"Are We Susceptible to Rowhammer? An End-to-End Methodology for Cloud Providers"**

*Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 2020.*

[Slides (pptx) (pdf)]

[Talk Video (17 minutes)]

## Are We Susceptible to Rowhammer?

### An End-to-End Methodology for Cloud Providers

Lucian Cojocar, Jeremie Kim<sup>§†</sup>, Minesh Patel<sup>§</sup>, Lillian Tsai<sup>‡</sup>,  
Stefan Saroiu, Alec Wolman, and Onur Mutlu<sup>§†</sup>  
Microsoft Research, <sup>§</sup>ETH Zürich, <sup>†</sup>CMU, <sup>‡</sup>MIT

# BlockHammer Solution in 2021

---

- A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu,

## **"BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows"**

*Proceedings of the 27th International Symposium on High-Performance Computer Architecture (HPCA)*, Virtual, February-March 2021.

[Slides (pptx) (pdf)]

[Short Talk Slides (pptx) (pdf)]

[Talk Video (22 minutes)]

[Short Talk Video (7 minutes)]

## **BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows**

A. Giray Yağlıkçı<sup>1</sup> Minesh Patel<sup>1</sup> Jeremie S. Kim<sup>1</sup> Roknoddin Azizi<sup>1</sup> Ataberk Olgun<sup>1</sup> Lois Orosa<sup>1</sup>  
Hasan Hassan<sup>1</sup> Jisung Park<sup>1</sup> Konstantinos Kanellopoulos<sup>1</sup> Taha Shahroodi<sup>1</sup> Saugata Ghose<sup>2</sup> Onur Mutlu<sup>1</sup>

<sup>1</sup>*ETH Zürich*

<sup>2</sup>*University of Illinois at Urbana-Champaign*

# Google's Recent RowHammer Attack (May 2021)

---



The latest news and insights from Google on security and safety on the Internet

---

## Introducing Half-Double: New hammering technique for DRAM Rowhammer bug

May 25, 2021

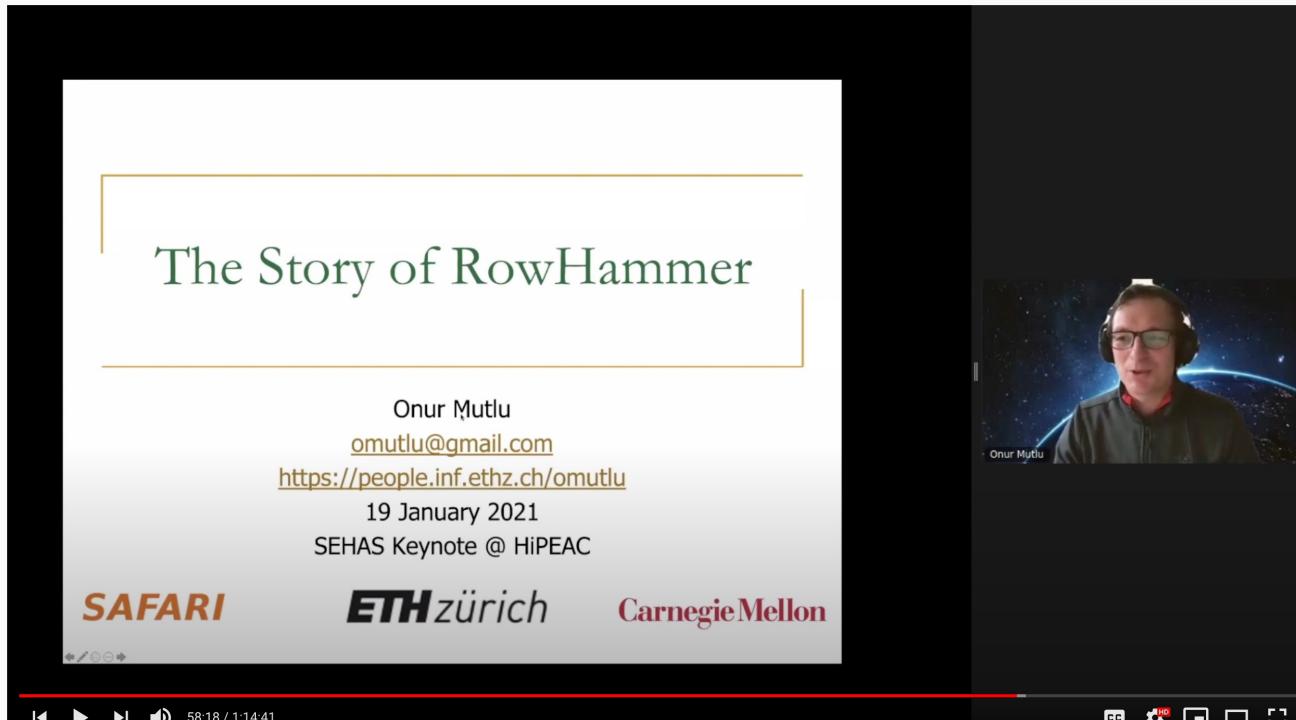
Research Team: Salman Qazi, Yoongu Kim, Nicolas Boichat, Eric Shiu & Mattias Nissler

Today, we are sharing details around our discovery of [Half-Double](#), a new Rowhammer technique that capitalizes on the worsening physics of some of the newer DRAM chips to alter the contents of memory.

Rowhammer is a DRAM vulnerability whereby repeated accesses to one address can tamper with the data stored at other addresses. Much like speculative execution vulnerabilities in CPUs, Rowhammer is a breach of the security guarantees made by the underlying hardware. As an electrical coupling phenomenon within the silicon itself, Rowhammer allows the potential bypass of hardware and software memory protection policies. This can allow untrusted code to break out of its sandbox and take full control of the system.

# The Story of RowHammer Lecture ...

- Onur Mutlu,  
[\*\*"The Story of RowHammer"\*\*](#)  
Keynote Talk at *Secure Hardware, Architectures, and Operating Systems Workshop (SeHAS)*, held with *HiPEAC 2021 Conference*, Virtual, 19 January 2021.  
[[Slides \(pptx\)](#) ([pdf](#))]  
[[Talk Video](#) (1 hr 15 minutes, with Q&A)]



The Story of Rowhammer - Secure Hardware, Architectures, and Operating Systems Keynote - Onur Mutlu

1,293 views • Premiered Feb 2, 2021

64 0 SHARE SAVE



Onur Mutlu Lectures  
13.9K subscribers

[ANALYTICS](#) [EDIT VIDEO](#)

# Detailed Lectures on RowHammer

---

- Computer Architecture, Fall 2020, Lecture 4b
  - RowHammer (ETH Zürich, Fall 2020)
  - <https://www.youtube.com/watch?v=KDy632z23UE&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=8>
- Computer Architecture, Fall 2020, Lecture 5a
  - RowHammer in 2020: TRRespass (ETH Zürich, Fall 2020)
  - [https://www.youtube.com/watch?v=pwRw7QqK\\_qA&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=9](https://www.youtube.com/watch?v=pwRw7QqK_qA&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=9)
- Computer Architecture, Fall 2020, Lecture 5b
  - RowHammer in 2020: Revisiting RowHammer (ETH Zürich, Fall 2020)
  - <https://www.youtube.com/watch?v=gR7XR-Eepcg&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=10>
- Computer Architecture, Fall 2020, Lecture 5c
  - Secure and Reliable Memory (ETH Zürich, Fall 2020)
  - <https://www.youtube.com/watch?v=HvswnsfG3oQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=11>

# Takeaway and Food for Thought

---

- If hardware is unreliable, higher-level security and protection mechanisms (as in virtual memory) may be compromised
- The root of security and trust is at the very low levels...
  - in the hardware itself
  - RowHammer, Spectre, Meltdown are recent key examples...
- What should we assume the hardware provides?
- How do we keep hardware reliable?
- How do we design secure hardware?
- How do we design secure hardware with high performance, high energy efficiency, low cost, convenient programming?

**Plenty of exciting and highly-relevant research questions**

# Some Issues in Virtual Memory

# Three Major Issues in Virtual Memory

---

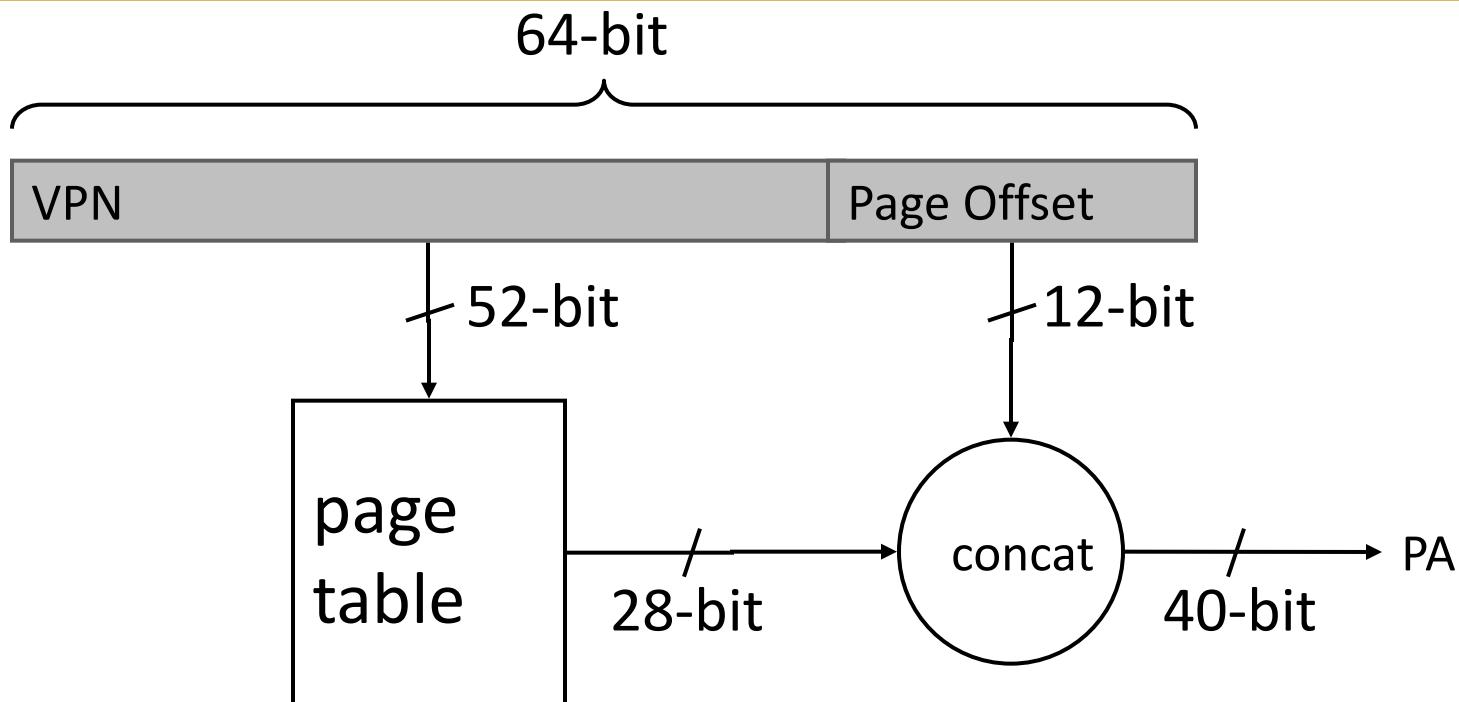
1. How large is the page table and how do we store and access it?
  2. How can we speed up translation & access control check?
  3. When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
    - What happens on a context switch?
    - How can you handle multiple page sizes?
    - ...

# Virtual Memory Issue I

---

- How large is the page table?
- Where do we store it?
  - ❑ In hardware?
  - ❑ In physical memory? (Where is the PTBR?)
  - ❑ In virtual memory? (Where is the PTBR?)
- How can we store it efficiently without requiring physical memory that can store all page tables?
  - ❑ Idea: multi-level page tables
  - ❑ Only the first-level page table has to be in physical memory
  - ❑ Remaining levels are in virtual memory (but get cached in physical memory when accessed)

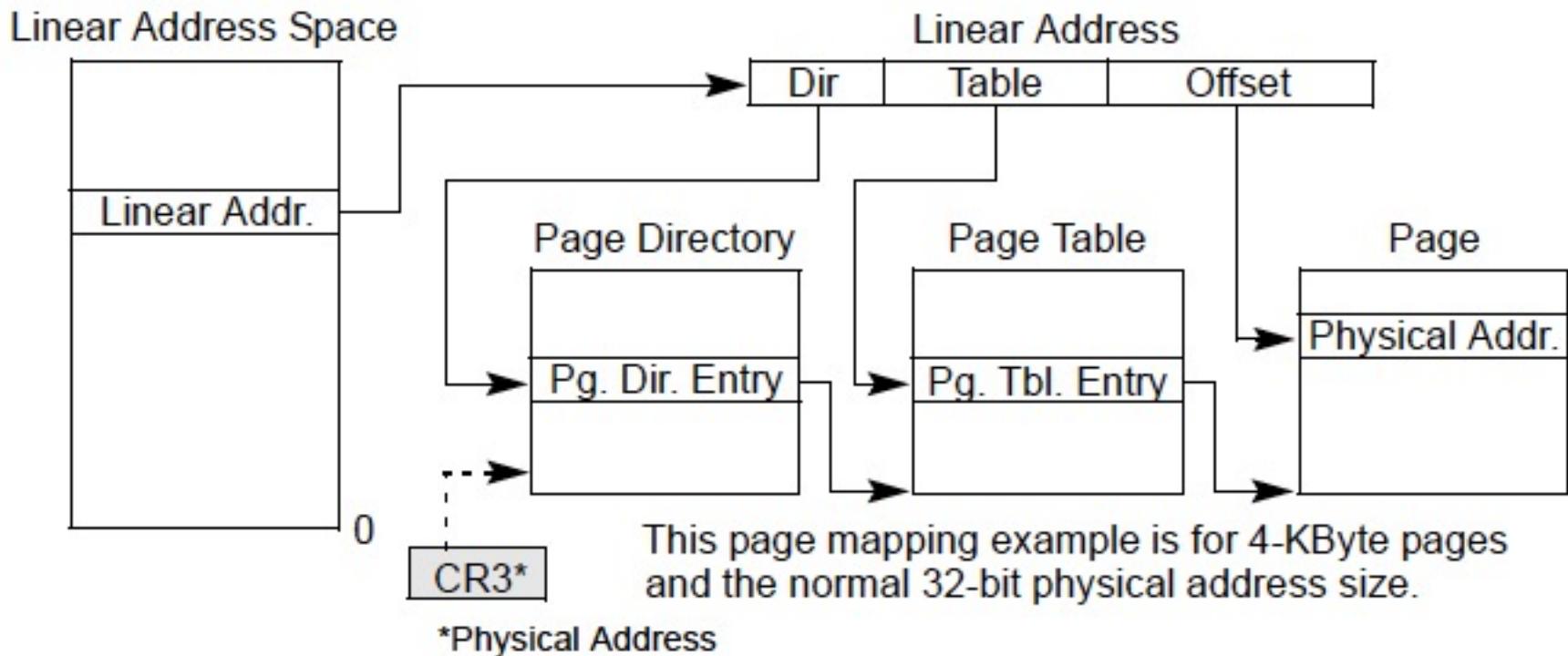
# Recall: Issue: Page Table Size



- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  - **$2^{52}$  entries x ~4 bytes  $\approx 2^{54}$  bytes**  
and that is for just one process!  
and the process may not be using the entire VM space!

# Recall: Solution: Multi-Level Page Tables

Example from the x86 architecture



# Page Table Access

---

- How do we access the Page Table?
- Page Table Base Register (CR3 in x86)
- Page Table Limit Register
- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception
- Page Table Base Register is part of a process's context
  - Just like PC, status registers, general purpose registers
  - Needs to be loaded when the process is context-switched in

# More on x86 Page Tables (I): Small Pages

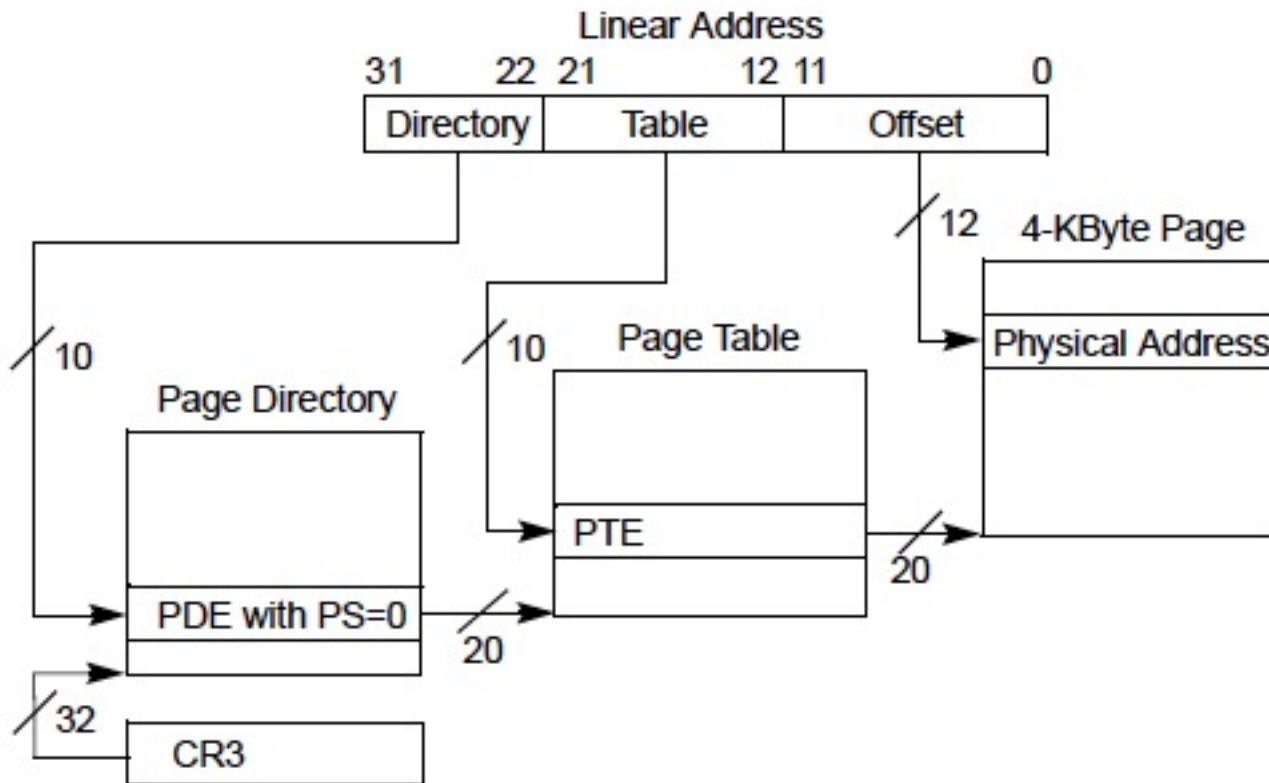


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

# More on x86 Page Tables (II): Large Pages

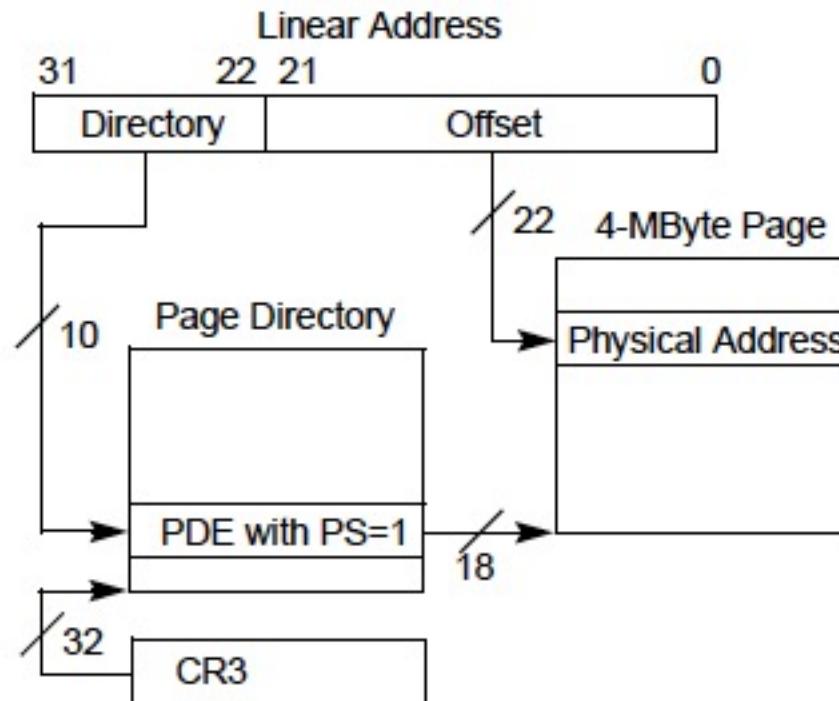


Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging

# x86 Page Table Entries

Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0												
Address of page directory <sup>1</sup>						Ignored			P C D	P W T	Ignored	CR3
Bits 31:22 of address of 2MB page frame	Reserved (must be 0)	Bits 39:32 of address <sup>2</sup>	P A T	Ignored	G	1	D	A	P C D	P W T	U / S W	PDE: 4MB page
Address of page table						Q	I g n	A	P C D	P W T	U / S W	PDE: page table
Ignored									0	PDE: not present		
Address of 4KB page frame						Ignored	G A T	P D	A C D	P W T	U / S W	PTE: 4KB page
Ignored									0	PTE: not present		

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

# x86 PTE (4KB page)

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0) <sup>1</sup>
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

# x86 Page Directory Entry (PDE)

---

**Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table**

<b>Bit Position(s)</b>	<b>Contents</b>
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CRO.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)

# Four-level Paging in x86-64

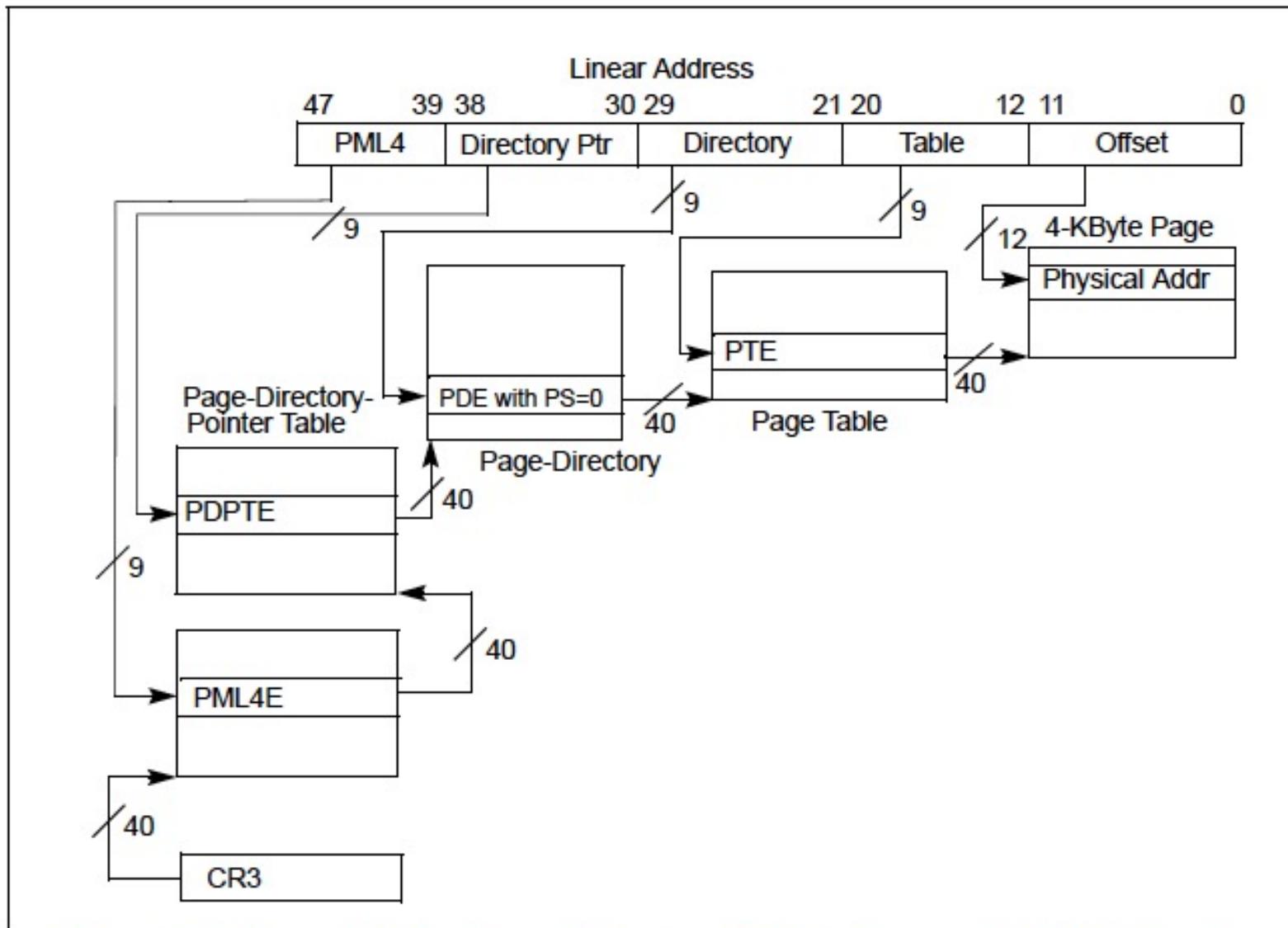


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

# Four-level Paging and Extended Physical Address Space in x86

---

A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32\_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.<sup>1</sup> Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with IA-32e paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:

# X86-64 Page Table Structure

6 3	6 2	6 1	6 0	5 9	5 8	5 7	5 6	5 5	5 4	5 3	5 2	5 1	M <sup>1</sup>	M-1	3 2	3 1	3 0	3 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	1 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9 8	7 7	6 6	5 5	4 4	3 3	2 2	1 1	0 0	1 0
Reserved <sup>2</sup>												Address of PML4 table (4-level paging) or PML5 table (5-level paging)												Ignored			P C W D T	P C W D T	Ign.	CR3																	
X D 3	Ignored			Rsvd.			Address of PML4 table												Ign.			R s v d n	I g n	A C W D T	P C W D T	U S W	R /S W	1	PML5E: present																		
Ignored																																						0	PML5E: not present								
X D 3	Ignored			Rsvd.			Address of page-directory-pointer table												Ign.			R s v d n	I g n	A C W D T	P C W D T	U S W	R /S W	1	PML4E: present																		
Ignored																																						0	PML4E: not present								
X D 3	Prot. Key <sup>4</sup>	Ignored			Rsvd.			Address of 1GB page frame				Reserved												P A T	Ign.	G 1	D A	P C W D T	U S W	R /S W	1	PDPTE: 1GB page															
X D 3	Ignored			Rsvd.			Address of page directory												Ign.			O I g n	A C W D T	P C W D T	U S W	R /S W	1	PDPTE: page directory																			
Ignored																																					0	PDTPE: not present									
X D 3	Prot. Key <sup>4</sup>	Ignored			Rsvd.			Address of 2MB page frame												Reserved			P A T	Ign.	G 1	D A	P C W D T	U S W	R /S W	1	PDE: 2MB page																
X D 3	Ignored			Rsvd.			Address of page table												Ign.			O I g n	A C W D T	P C W D T	U S W	R /S W	1	PDE: page table																			
Ignored																																				0	PDE: not present										
X D 3	Prot. Key <sup>4</sup>	Ignored			Rsvd.			Address of 4KB page frame												Ign.			P A T	G 1	D A	P C W D T	U S W	R /S W	1	PTE: 4KB page																	
Ignored																																				0	PTE: not present										

Figure 4-11. Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging

# X86-64 Page Table: Accessing 4KB pages

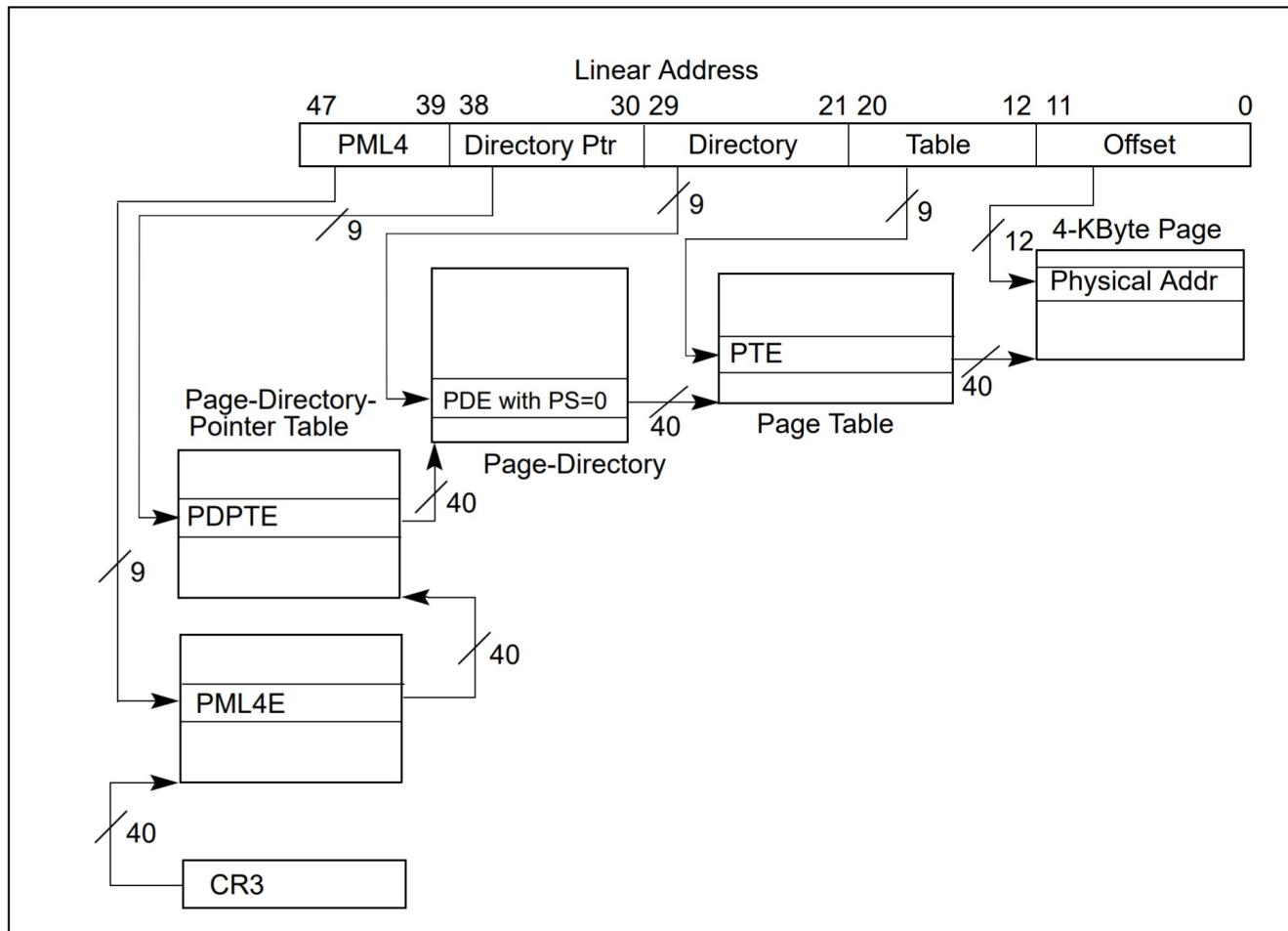


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

# X86-64 Page Table: Accessing 2MB pages

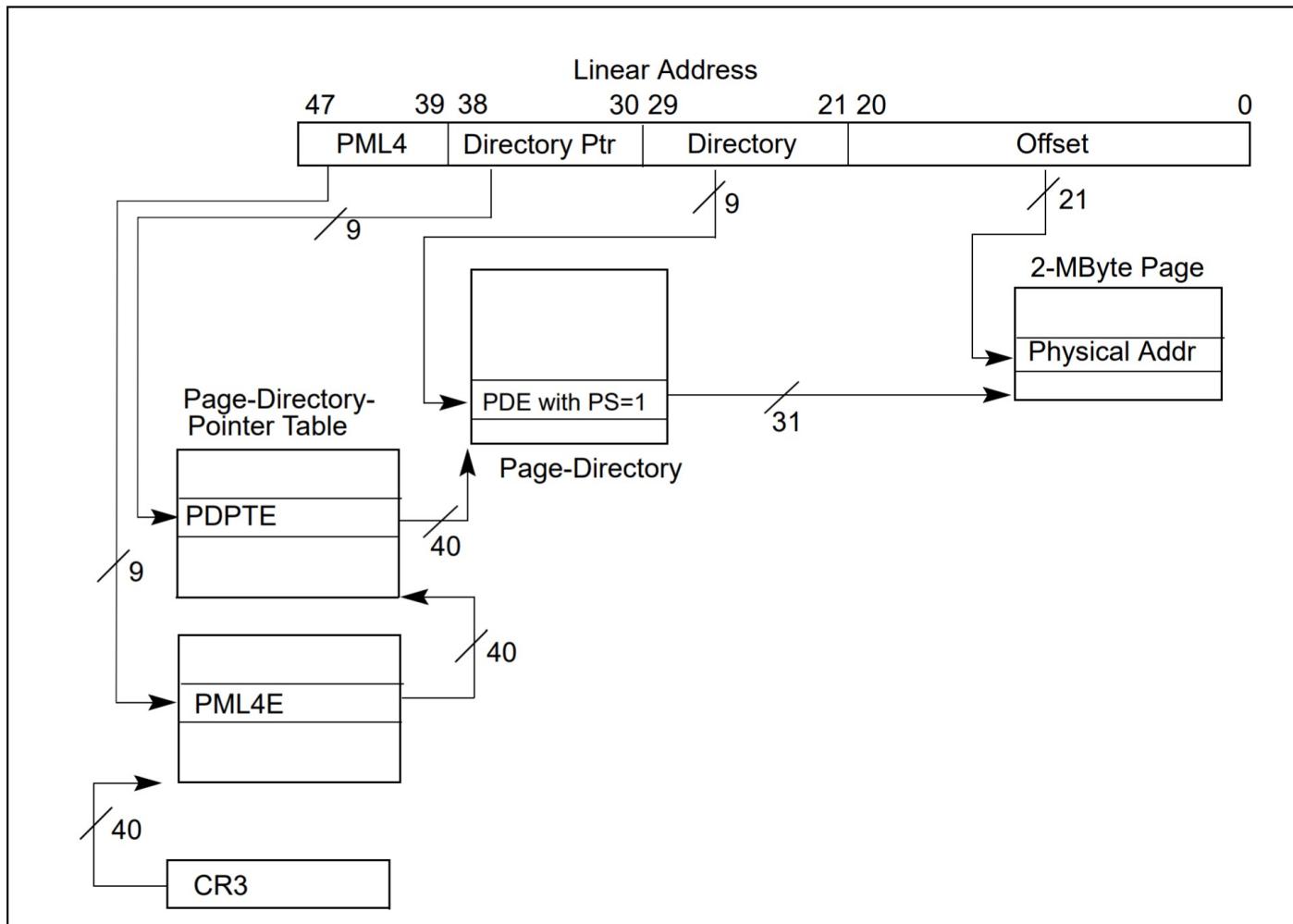


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

# X86-64 Page Table: Accessing 1GB pages

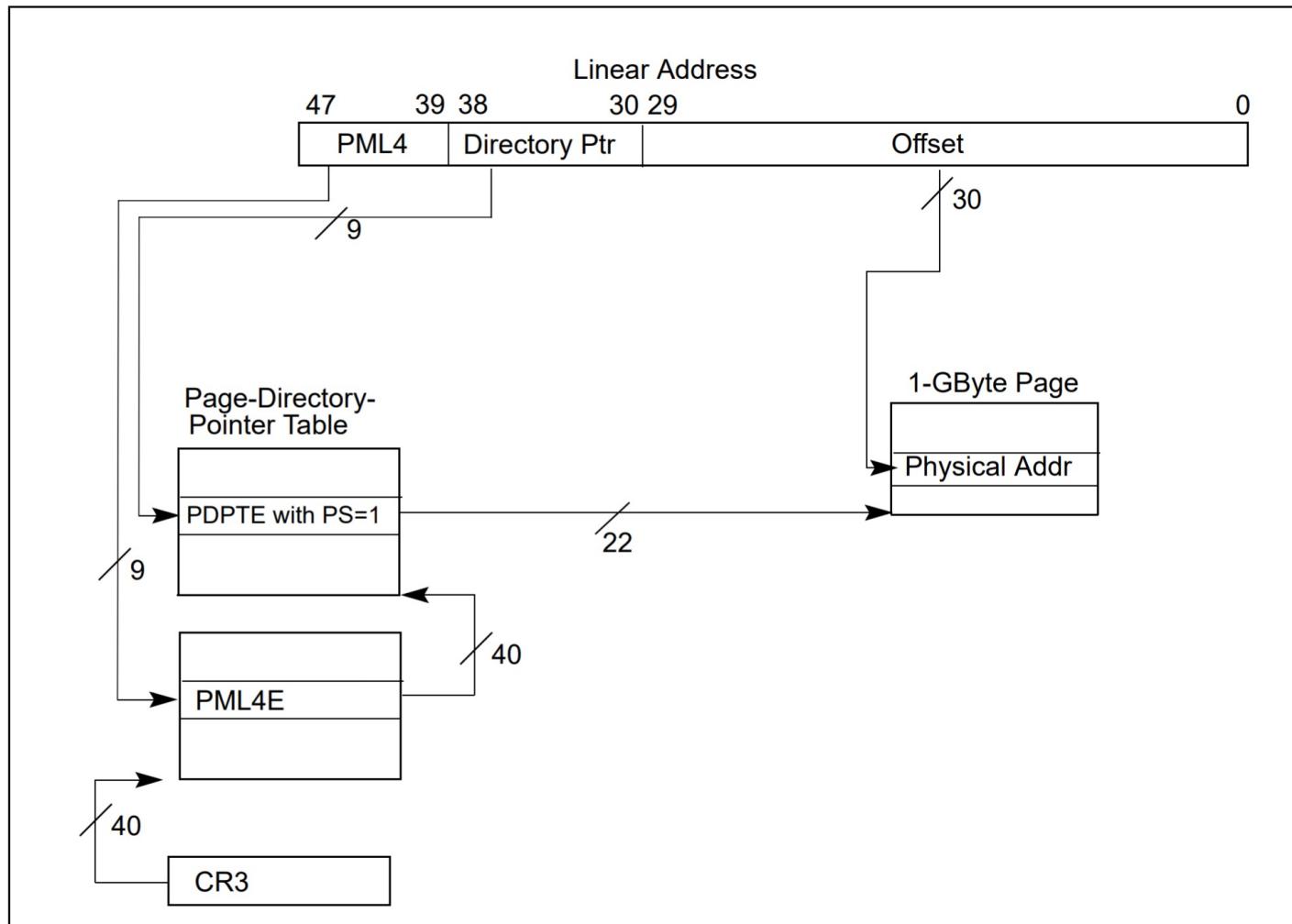


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

# Three Major Issues in Virtual Memory

---

1. How large is the page table and how do we store and access it?
  2. How can we speed up translation & access control check?
  3. When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
    - What happens on a context switch?
    - How can you handle multiple page sizes?
    - ...

# Recall: Translation Lookaside Buffer (TLB)

---

- Idea: Cache the page table entries (PTEs) in a hardware structure in the processor to speed up address translation
- Translation lookaside buffer (TLB)
  - Small cache of most recently used translations (PTEs)
  - Reduces number of memory accesses required for *most* instruction fetches and loads/stores to only one

# Virtual Memory Issue II

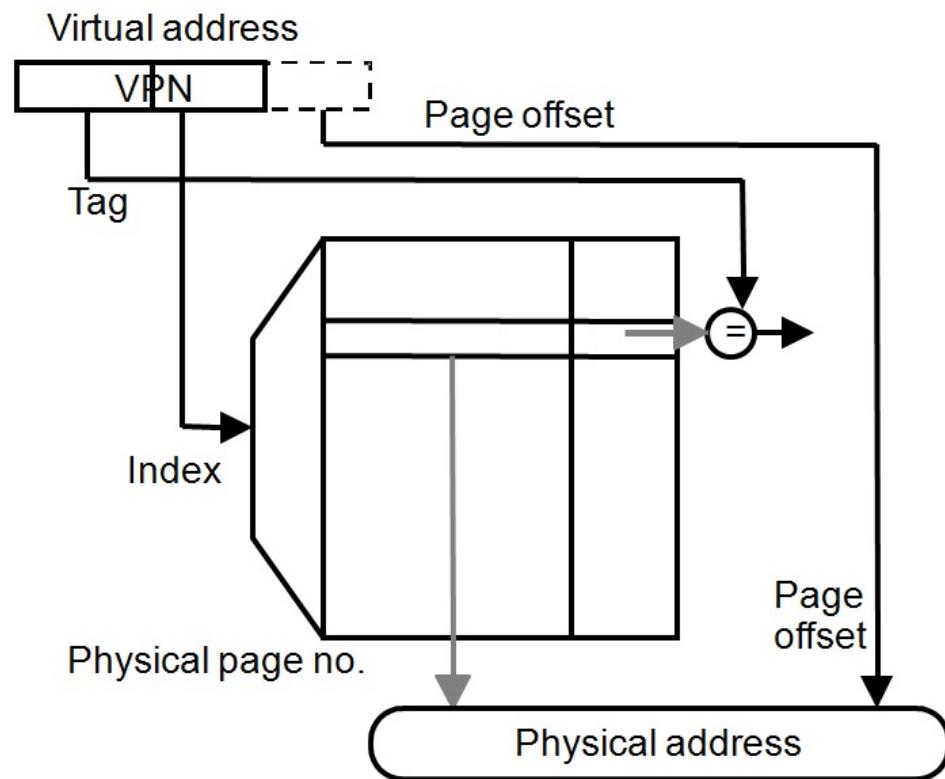
---

- How fast is the address translation?
  - How can we make it fast?
- Idea: Use a hardware structure that caches PTEs → Translation Lookaside Buffer (TLB)
- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?
- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?

# Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference
- **Index** = lower bits of VPN (virtual page #)
- **Tag** = unused bits of VPN + process ID
- **Data** = a page-table entry
- **Status** = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.



# Handling TLB Misses

---

- The TLB is small; it cannot hold all PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called **walking** the page table
    - Large performance penalty
- Who handles TLB misses? Hardware or software?

# Handling TLB Misses (II)

---

- Approach #1. **Hardware-Managed** (e.g., x86)
  - The hardware does the **page walk**
  - The hardware fetches the PTE and inserts it into the TLB
    - If the TLB is full, the entry **replaces** another entry
  - Done transparently to system software
  
- Approach #2. **Software-Managed** (e.g., MIPS)
  - The hardware raises an exception
  - The operating system does the **page walk**
  - The operating system fetches the PTE
  - The operating system inserts/evicts entries in the TLB

# Handling TLB Misses (III)

---

## ■ Hardware-Managed TLB

- Pro: No exception on TLB miss. Instruction just stalls
- Pro: Independent instructions may continue
- Pro: No extra instructions/data brought into caches.
- Con: Page directory/table organization is etched into the system: OS has little flexibility in deciding these

## ■ Software-Managed TLB

- Pro: The OS can define page table organization
- Pro: More sophisticated TLB replacement policies are possible
- Con: Need to generate an exception → performance overhead due to pipeline flush, exception handler execution, extra instructions brought to caches

# Three Major Issues in Virtual Memory

---

1. How large is the page table and how do we store and access it?
  2. How can we speed up translation & access control check?
  3. When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
    - What happens on a context switch?
    - How can you handle multiple page sizes?
    - ...

# Teaser: Virtual Memory Issue III

---

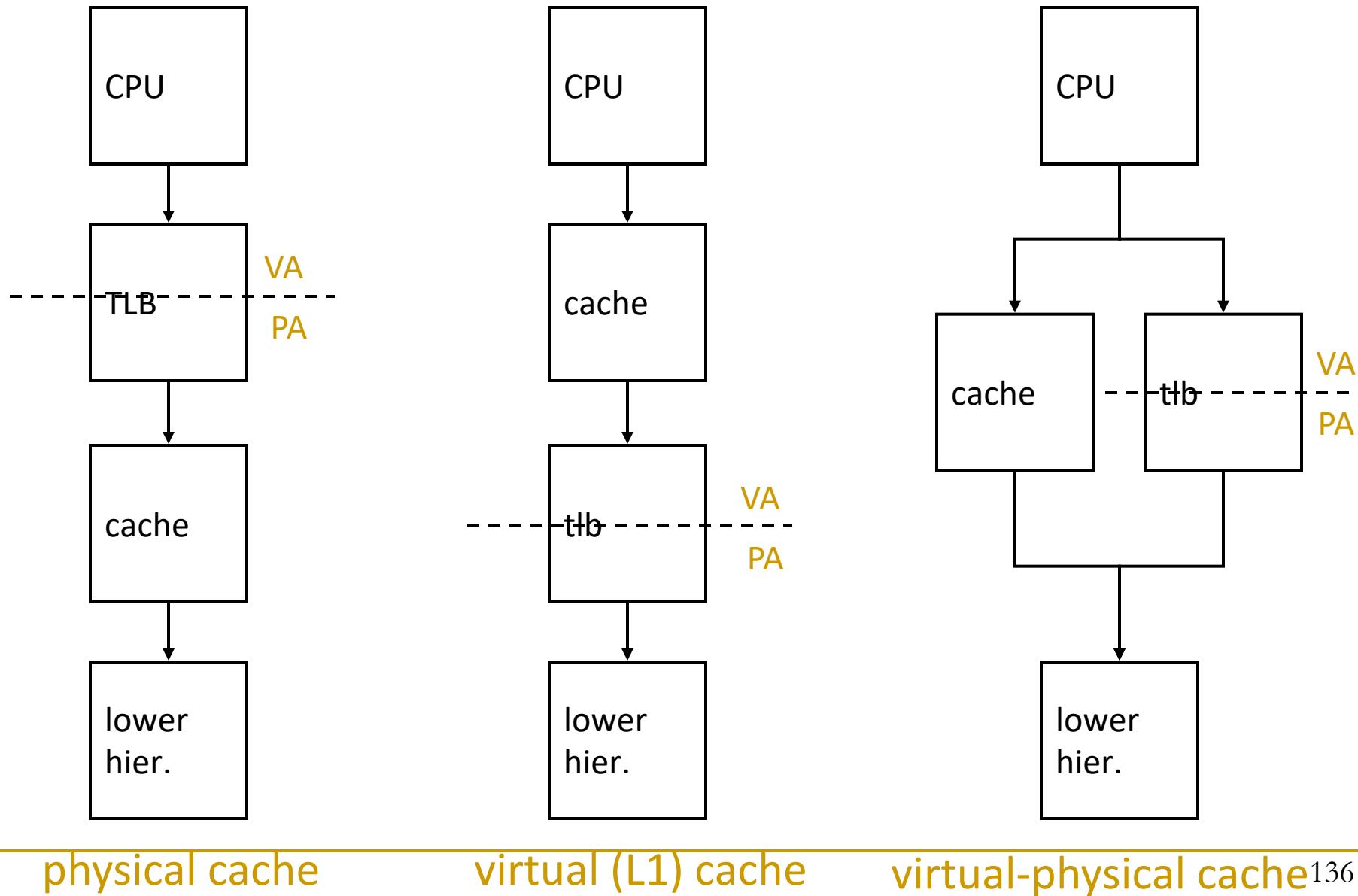
- When do we do the address translation?
  - Before or after accessing the L1 cache?

# Address Translation and Caching

---

- When do we do the address translation?
  - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
  - **Virtual versus physical cache**
- What are the issues with a virtually addressed cache?
- **Synonym problem:**
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

# Cache-VM Interaction



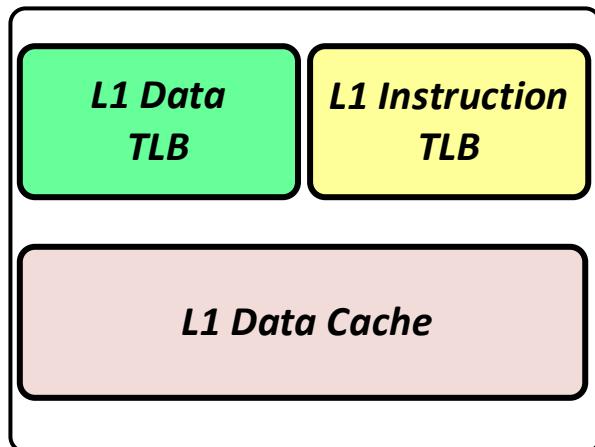
# A Modern Example

## Virtual Memory System

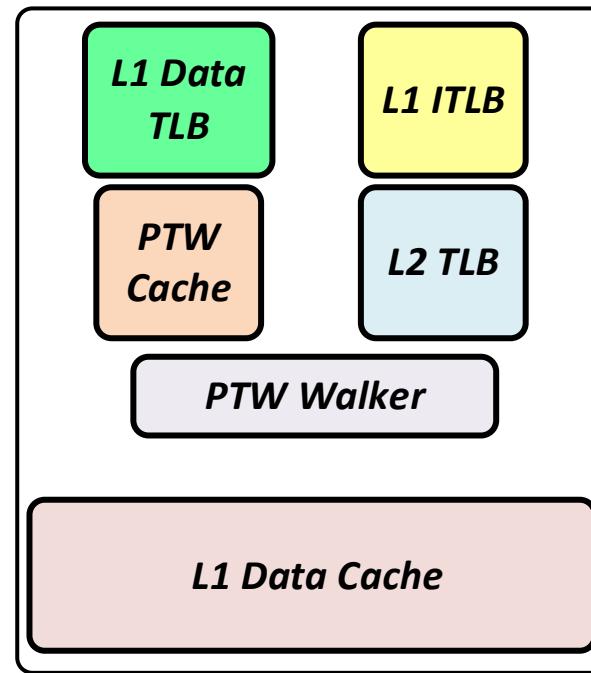
# Evolution of Address Translation

---

## Conventional Address Translation



## Modern Address Translation

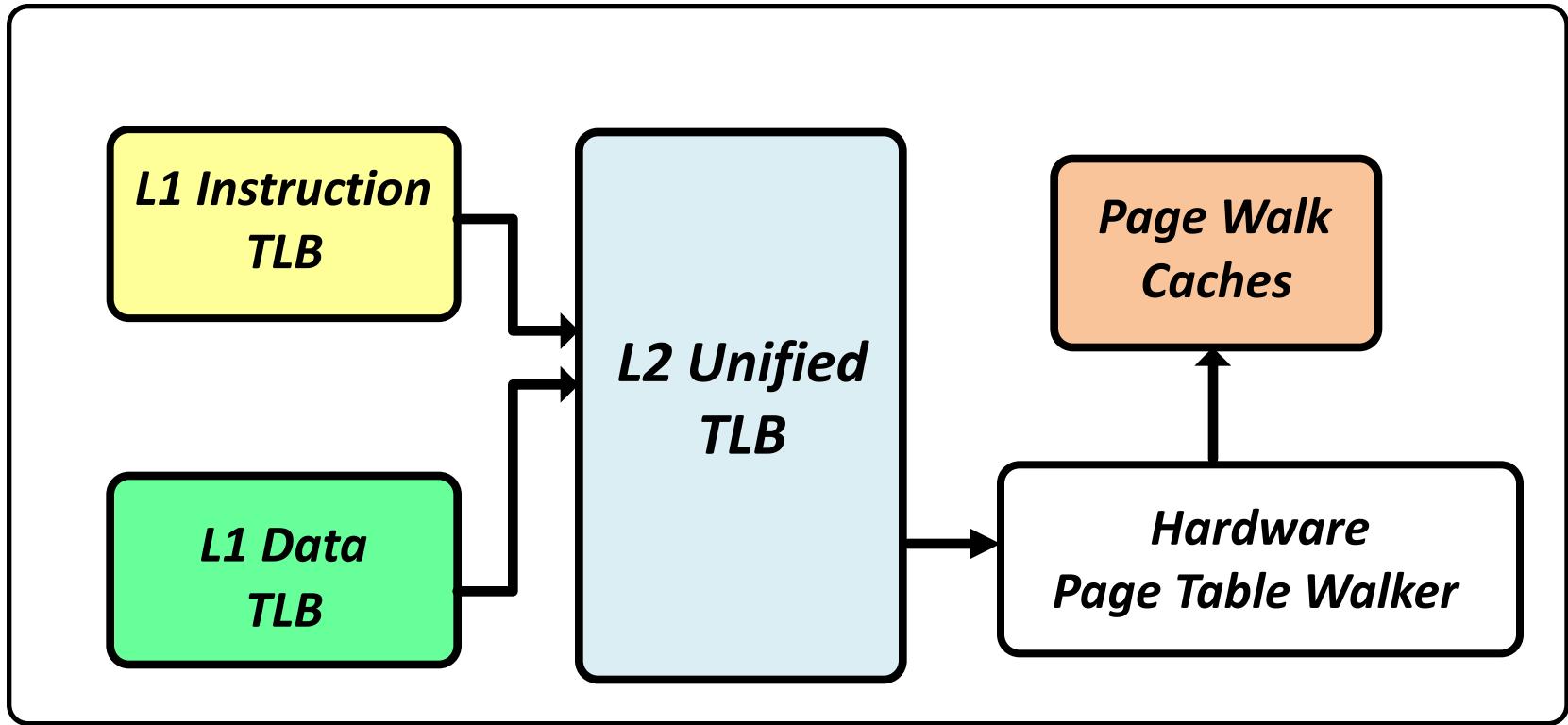


# Memory Management Unit

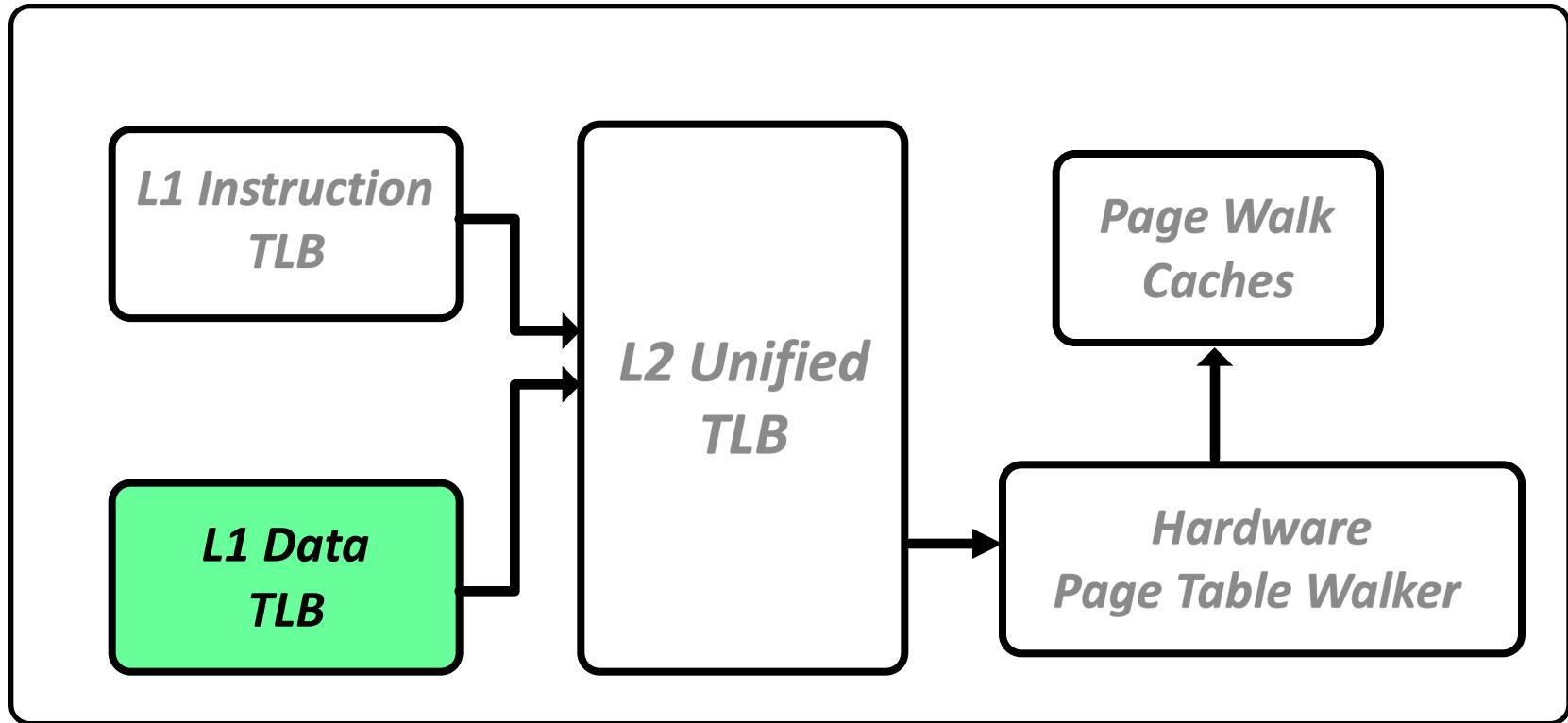
---

- The **Memory Management Unit (MMU)** is a per-core component, responsible for resolving address translation requests
- MMU typically has three key components:
  - **Translation Lookaside Buffers** that cache recently accessed virtual-to-physical translations
  - **Page Table Walk Caches** that offer fast access to the L4,L3,L2 levels of the Page table
  - **Hardware Page Table Walker** that sequentially accesses the different levels of the Page Table to fetch the PTE

# Intel Skylake: MMU



# Intel Skylake: L1 Data TLB

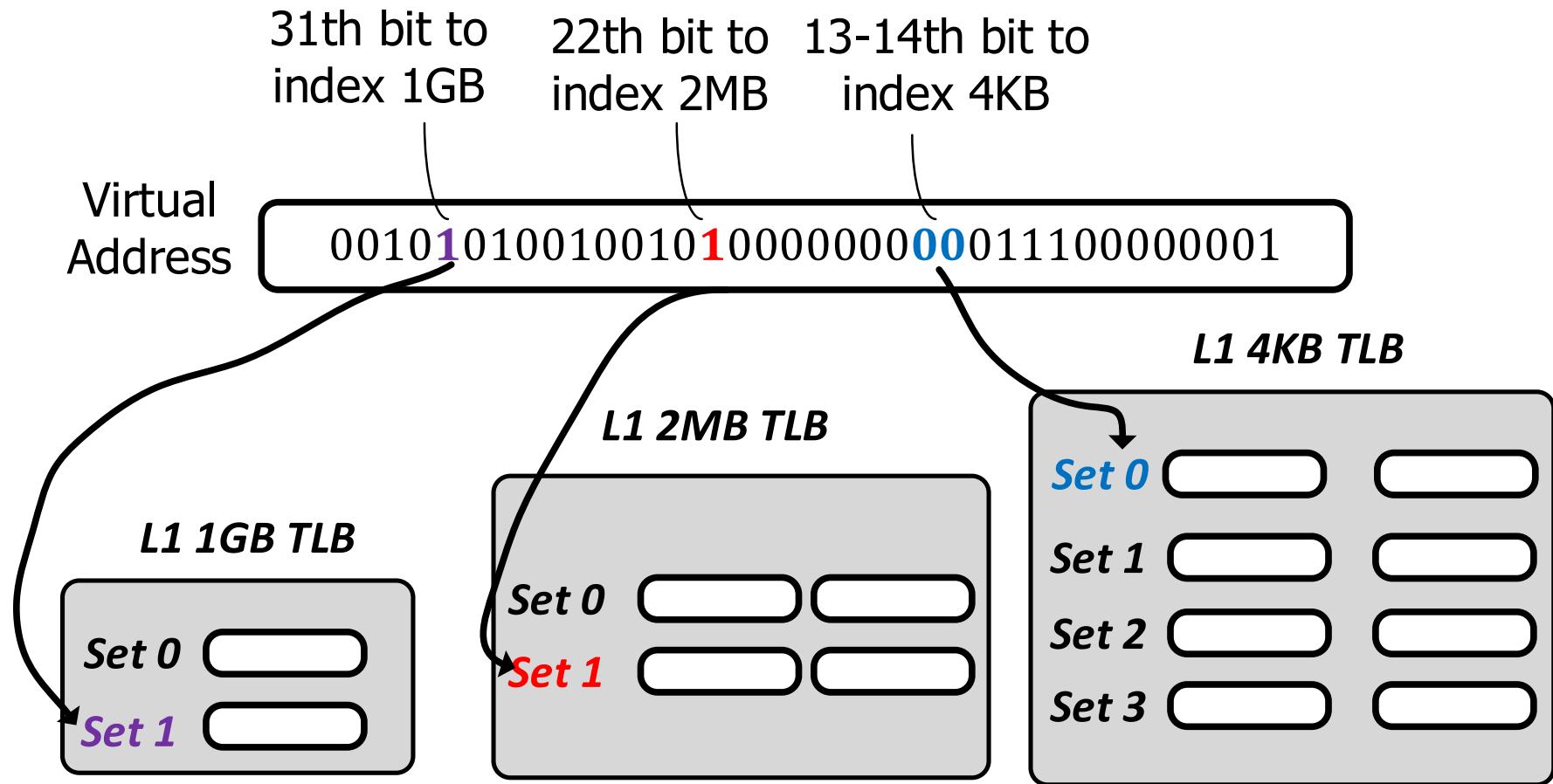


# Intel Skylake: L1 Data TLB

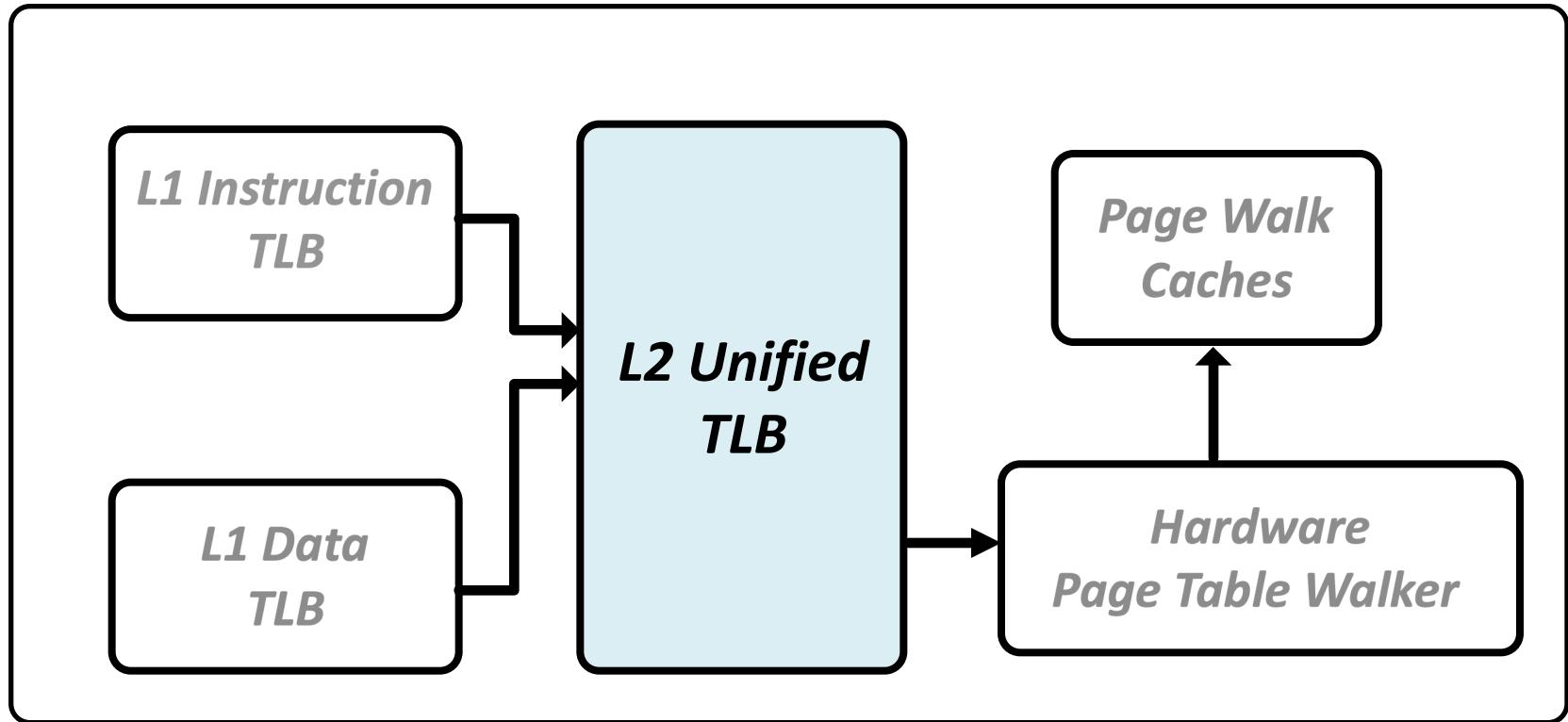
---

- Separate L1 Data TLB structures for **4KB**, **2MB**, and **1GB** pages
- L1 DTLB
  - **4KB**: 64-entry, 4-way, 1 cycle access, 9 cycle miss
  - **2MB**: 32-entry, 4-way, 1 cycle access, 9 cycle miss
  - **1GB**: 4 entry, fully-associative, 1 cycle access, 9 cycle miss
- Virtual-to-physical mappings are inserted in the corresponding TLB after a TLB miss
- During a translation request, all three L1 TLBs are looked up in parallel

# L1 Data TLB: Example



# Intel Skylake: L2 Unified TLB



# Intel Skylake: L2 Unified TLB

---

- L2 Unified TLB caches translations for both instruction and data and is private per individual core
- 2 Separate L2 TLB structures for 4KB/2MB and 1GB pages
- L2 TLB
  - **4KB/2MB**: 1536-entry, 12-way, 14 cycle access, 9 cycle miss
  - **1GB**: 16-entry, 4-way, 1 cycle access, 9 cycle miss penalty
- Challenge: How can the L2 TLB support both 4KB and 2MB pages using a single structure?  
(Not enough publicly available information for Intel Skylake)

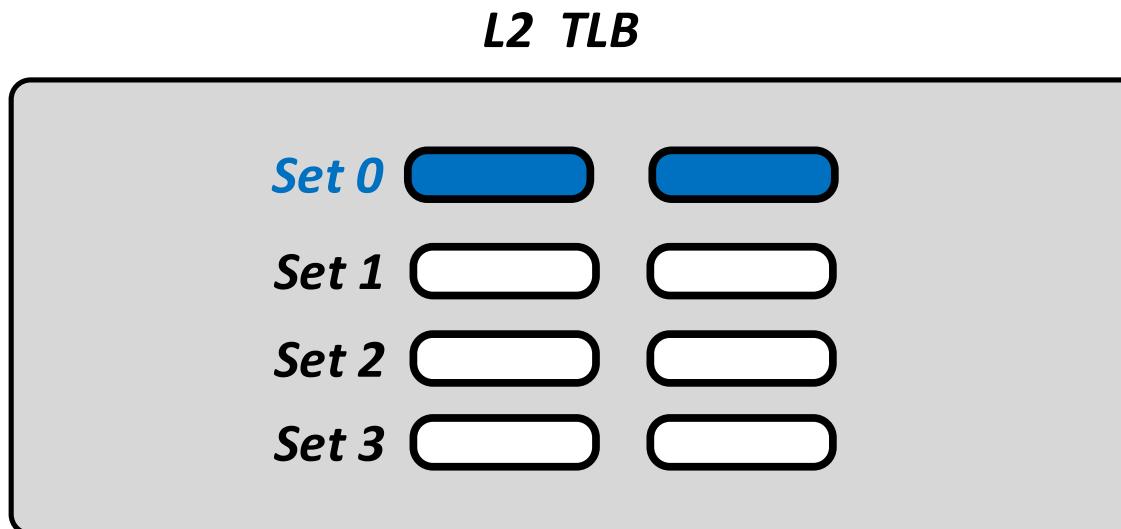
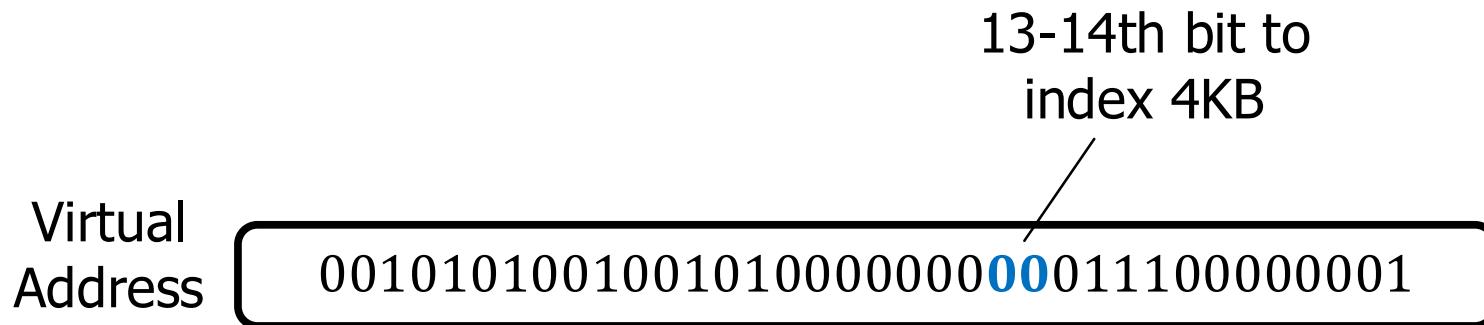
# L2 Unified TLB: Accessing the TLB

---

- The 4KB/2MB structure of the L2 TLB is **probed in 2 steps**
- **Step 1:** Assume the page size is **4KB**, calculate the index bits and access the L2 TLB. If the tag matches, it is a hit. If the tag does not match, go to Step 2.
- **Step 2:** Assume the page size is **2MB**, **re-calculate** the index and access the L2 TLB. If the tag matches, it is a hit. If the tag does not match, it is an L2 TLB miss.
- **General algorithm:**  
Re-calculate index and probe TLB for all remaining page sizes

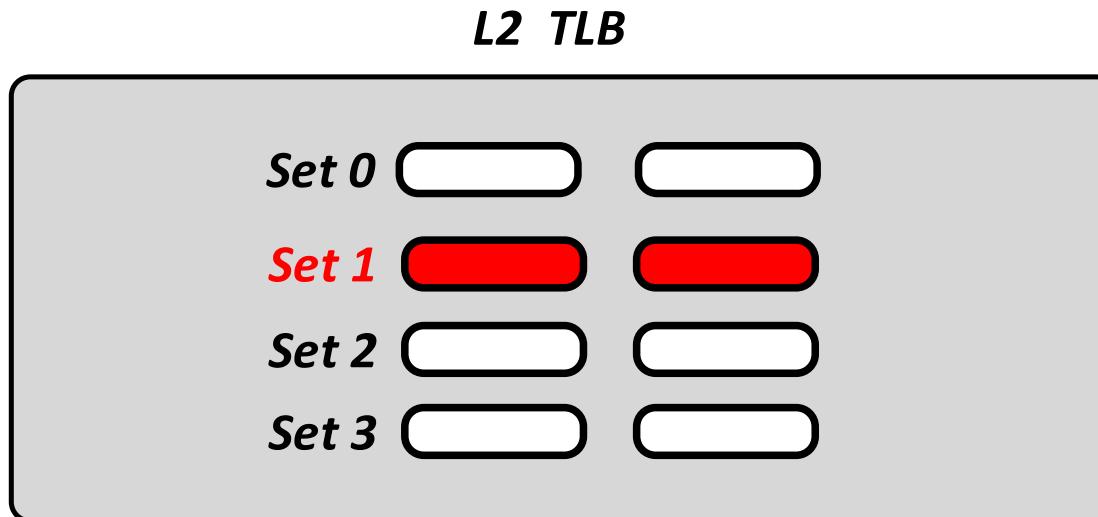
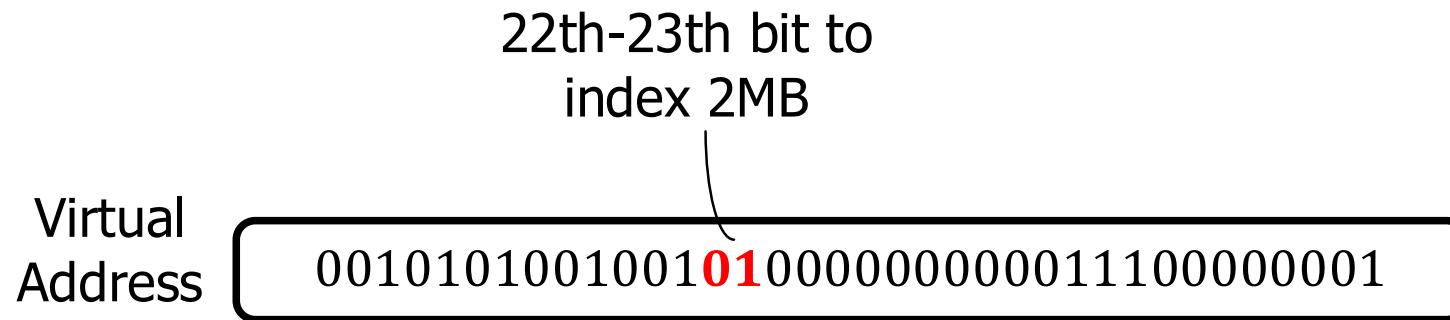
# Step 1: Calculate index for 4KB

---



# Step 2: Re-calculate index for 2MB

---

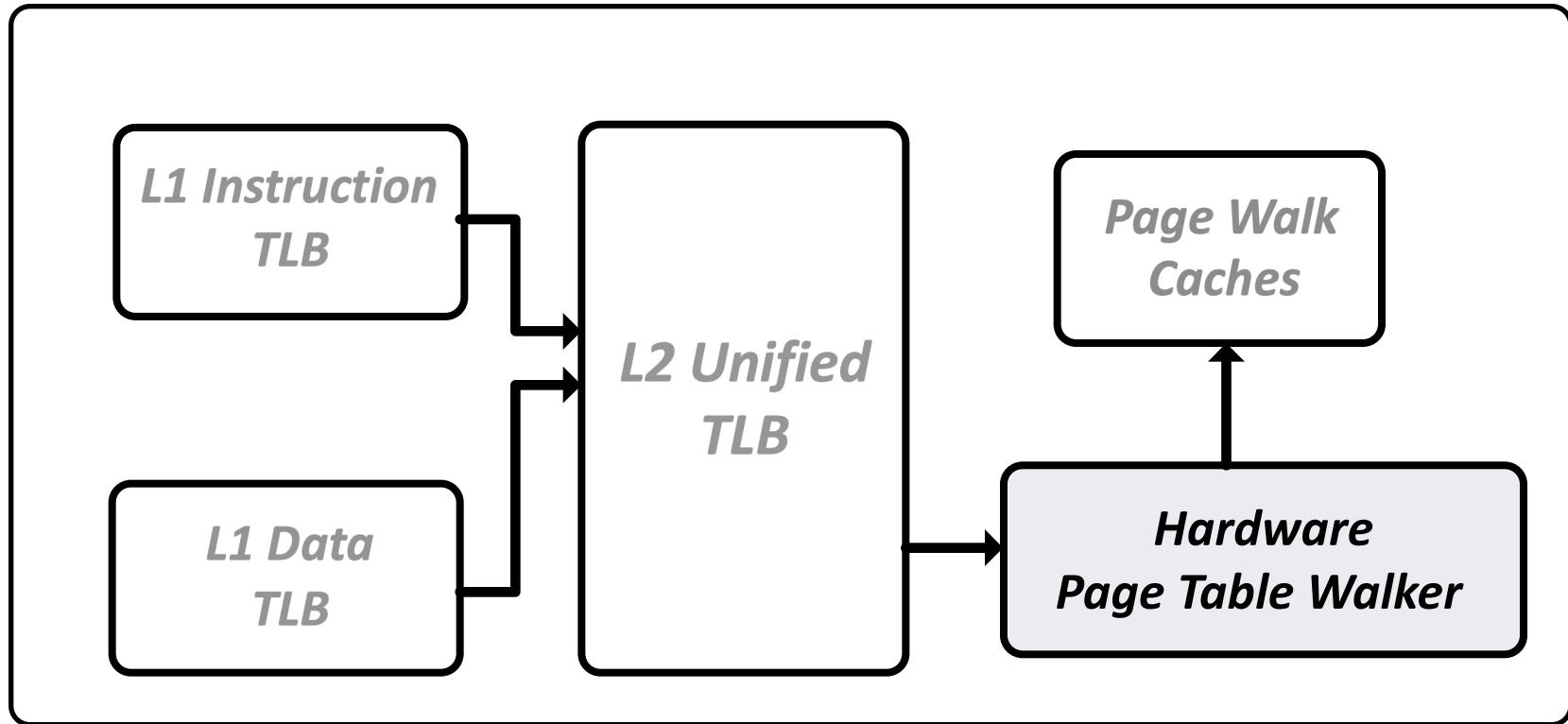


# L2 TLB: N-Step Index Re-Calculation

---

- Pros:
  - + Simple and practical implementation
- Cons:
  - Varying L2 TLB hit latency (faster for 4KB, slower for 2MB)
  - Slower identification of L2 TLB Miss as all page sizes need to be tested
- Potential Optimizations:
  - (1) Parallel Lookups: Lookup for 4KB and 2MB in parallel
  - (2) Page Size Prediction: Predict the probing order

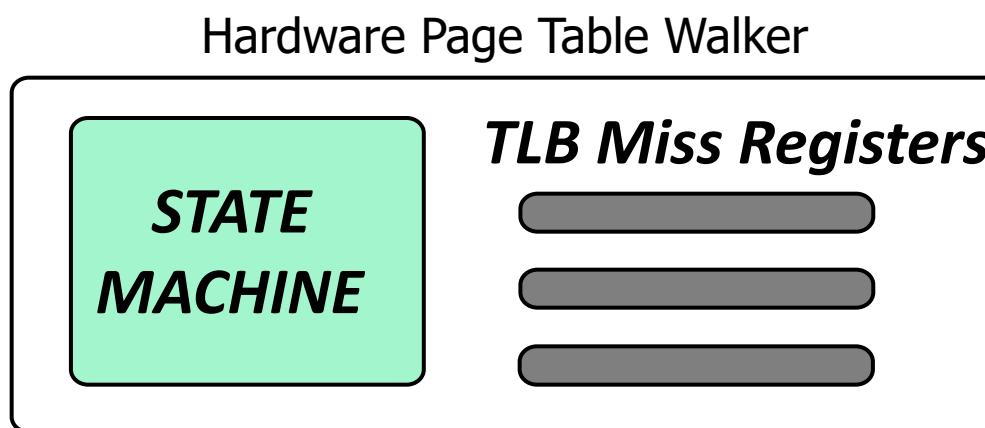
# Hardware Page Table Walker



# Hardware Page Table Walker (I)

---

- The MMU employs a per-core hardware component that accesses the page table to avoid expensive context switches
- HW PTW consists of 2 components:
  - A state machine that is designed to be aware of the architecture's page table structure
  - Registers that keep track of outstanding TLB misses



# Hardware Page Table Walker (II)

---

- Pros:
  - + Avoid the need for context switches on TLB misses
  - + Overlap TLB misses with useful computation
  - + Supports concurrent TLB misses
- Cons:
  - Increase in area and power overheads
  - Limited flexibility compared to software page table walk

# Hardware Page Table Walker (III)

- PTW accesses the CR3 register that maintains information about the physical address of the root of the page table (PML4)
- PTW concatenates the content of CR3 with the first 9 bits of the virtual address

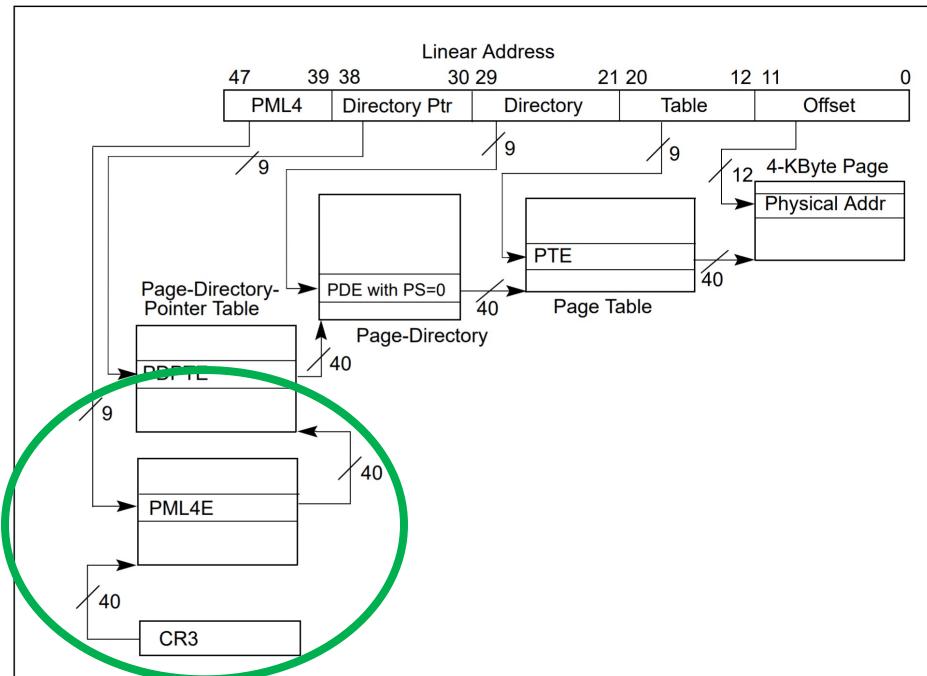


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

# Hardware Page Table Walker (IV)

- Hardware PTWs allow overlapping TLB misses with useful computation

## Software PTW

VPN = 1

*LOAD A*

*TLB Miss*

*Context Switch – TLB Miss Handler*

VPN = 5

*LOAD B*

*TLB Hit*

Saved Cycles

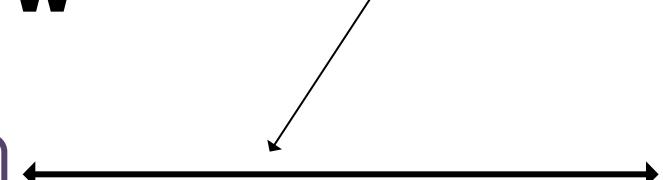
## Hardware PTW

VPN = 1

*LOAD A*

*TLB Miss*

*Page Table Walk*



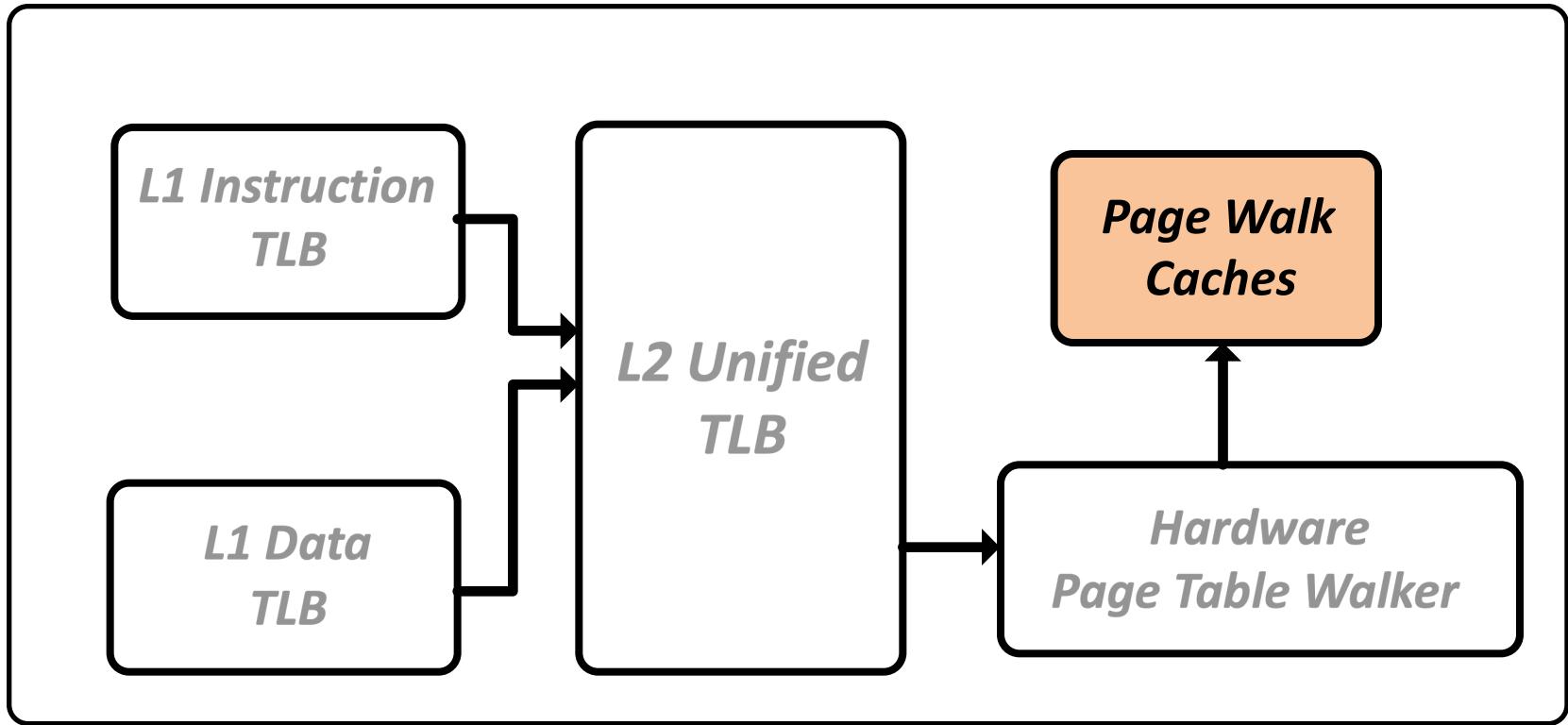
*LOAD B*

*TLB Hit*

VPN = 5

# Page Walk Caches

---

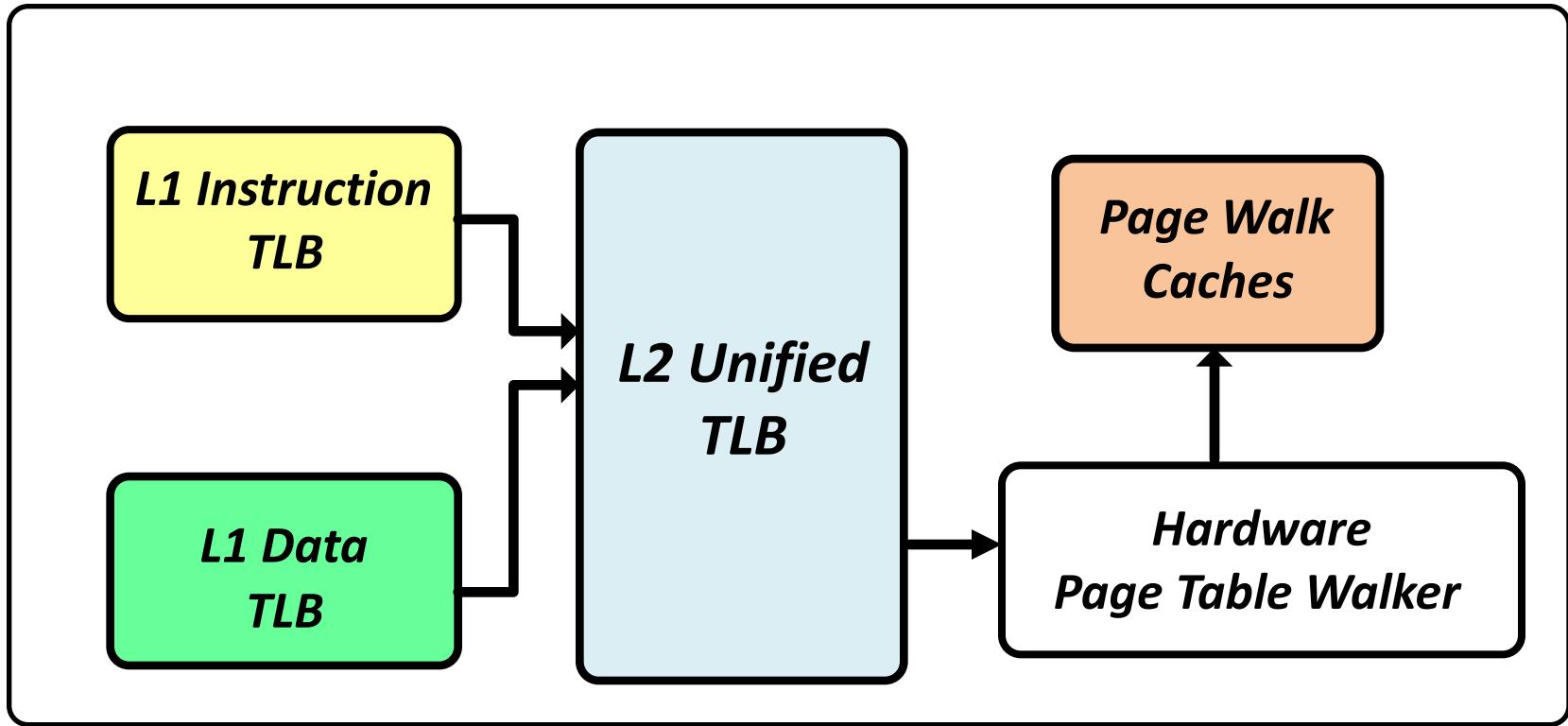


# Page Walk Caches

---

- Page Walk Caches are low-latency caches which offer faster access to the page table compared to accessing the cache hierarchy for every page table access
- Page Walk Caches store translations from non-leaf entries of the page table to accelerate page table walks

# Intel Skylake: MMU



# Virtual Memory Summary

# Virtual Memory Summary

---

- Virtual memory gives the illusion of “infinite” capacity
  - A subset of virtual pages are located in physical memory
  - A **page table** maps virtual pages to physical pages – this is called address translation
  - A **TLB** speeds up address translation
  - **Multi-level page tables** keep the page table size in check
  - Using different page tables for different programs provides **memory protection**
-

# Virtual Memory: Parting Thoughts

---

- VM is one of the most successful examples of
  - architectural support for programmers
  - how to partition work between hardware and software
  - hardware/software cooperation
  - programmer/architect tradeoff
- Going forward: How does virtual memory scale into the future? Four key trends:
  - Increasing, huge physical memory sizes
  - Hybrid physical memory systems (DRAM + NVM + SSD)
  - Many accelerators in the system addressing physical memory
  - Virtualized systems (hypervisors, software virtualization, local and remote memories)

# Rethinking Virtual Memory

---

Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo Francisco de Oliveira Jr., Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu,  
**"The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework"**

*Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, Virtual, June 2020.

[[Slides \(pptx\)](#) ([pdf](#))]

[[Lightning Talk Slides \(pptx\)](#) ([pdf](#))]

[[ARM Research Summit Poster \(pptx\)](#) ([pdf](#))]

[[Talk Video](#) (26 minutes)]

[[Lightning Talk Video](#) (3 minutes)]

[[Lecture Video](#) (43 minutes)]

## The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework

Nastaran Hajinazar<sup>\*†</sup> Pratyush Patel<sup>✉</sup> Minesh Patel<sup>\*</sup> Konstantinos Kanellopoulos<sup>\*</sup> Saugata Ghose<sup>‡</sup>  
Rachata Ausavarungnirun<sup>○</sup> Geraldo F. Oliveira<sup>\*</sup> Jonathan Appavoo<sup>◊</sup> Vivek Seshadri<sup>▽</sup> Onur Mutlu<sup>\*‡</sup>

<sup>\*</sup>*ETH Zürich*   <sup>†</sup>*Simon Fraser University*   <sup>✉</sup>*University of Washington*   <sup>‡</sup>*Carnegie Mellon University*

<sup>○</sup>*King Mongkut's University of Technology North Bangkok*   <sup>◊</sup>*Boston University*   <sup>▽</sup>*Microsoft Research India*

# Lectures on Virtual Memory

## Challenges

- Three examples of the **challenges** in adapting conventional virtual memory frameworks for increasingly-diverse systems:
  - Requiring a **rigid page table structure**
  - High address **translation overhead** in virtual machines
  - **Inefficient** heterogeneous memory **management**



SAFARI 9:22 / 42:44

12

CC G S M

ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 12c: The Virtual Block Interface (ETH Zürich, Fall 2020)

726 views • Oct 31, 2020

16 0 SHARE SAVE ...



Onur Mutlu Lectures  
16.5K subscribers

ANALYTICS

EDIT VIDEO

# Lectures on Virtual Memory

The image shows a YouTube video player interface. The main content is a presentation slide titled "Some Solutions to the Synonym Problem". The slide lists three solutions:

- Limit cache size to (page size times associativity)
  - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - make sure  $\text{index(VA)} = \text{index(PA)}$
  - Called page coloring
  - Used in many SPARC processors

At the bottom of the slide, there are navigation icons for a presentation slide. The video player interface includes a progress bar at 1:43:45 / 1:44:49, a red scrub bar, and standard YouTube controls for play, volume, and settings.

Lecture 20. Virtual Memory - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

22,313 views • Mar 7, 2015

139 5 SHARE SAVE ...



Carnegie Mellon Computer Architecture  
23.3K subscribers

SUBSCRIBED



# Lectures on Virtual Memory

---

- Computer Architecture, Spring 2015, Lecture 20
  - Virtual Memory (CMU, Spring 2015)
  - <https://www.youtube.com/watch?v=2RhGMpY18zw&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=22>
- Computer Architecture, Fall 2020, Lecture 12c
  - The Virtual Block Interface (ETH, Fall 2020)
  - <https://www.youtube.com/watch?v=PPR7YrBi7IQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=24>

# Backup Slides

# More on Issues in Virtual Memory

# Virtual Memory and Cache Interaction

# Address Translation and Caching

---

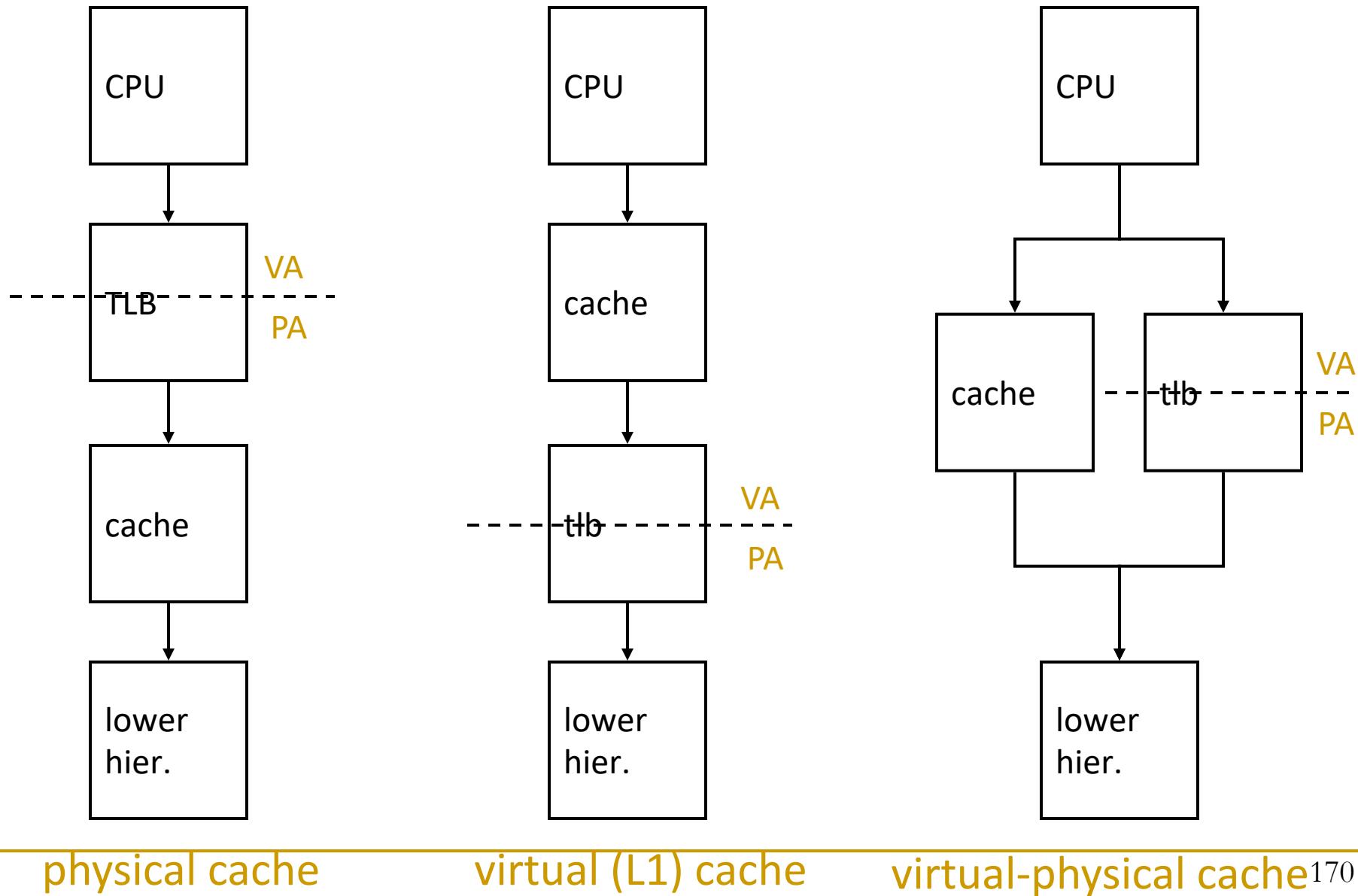
- When do we do the address translation?
  - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
  - **Virtual versus physical cache**
- What are the issues with a virtually addressed cache?
- **Synonym problem:**
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data

# Homonyms and Synonyms

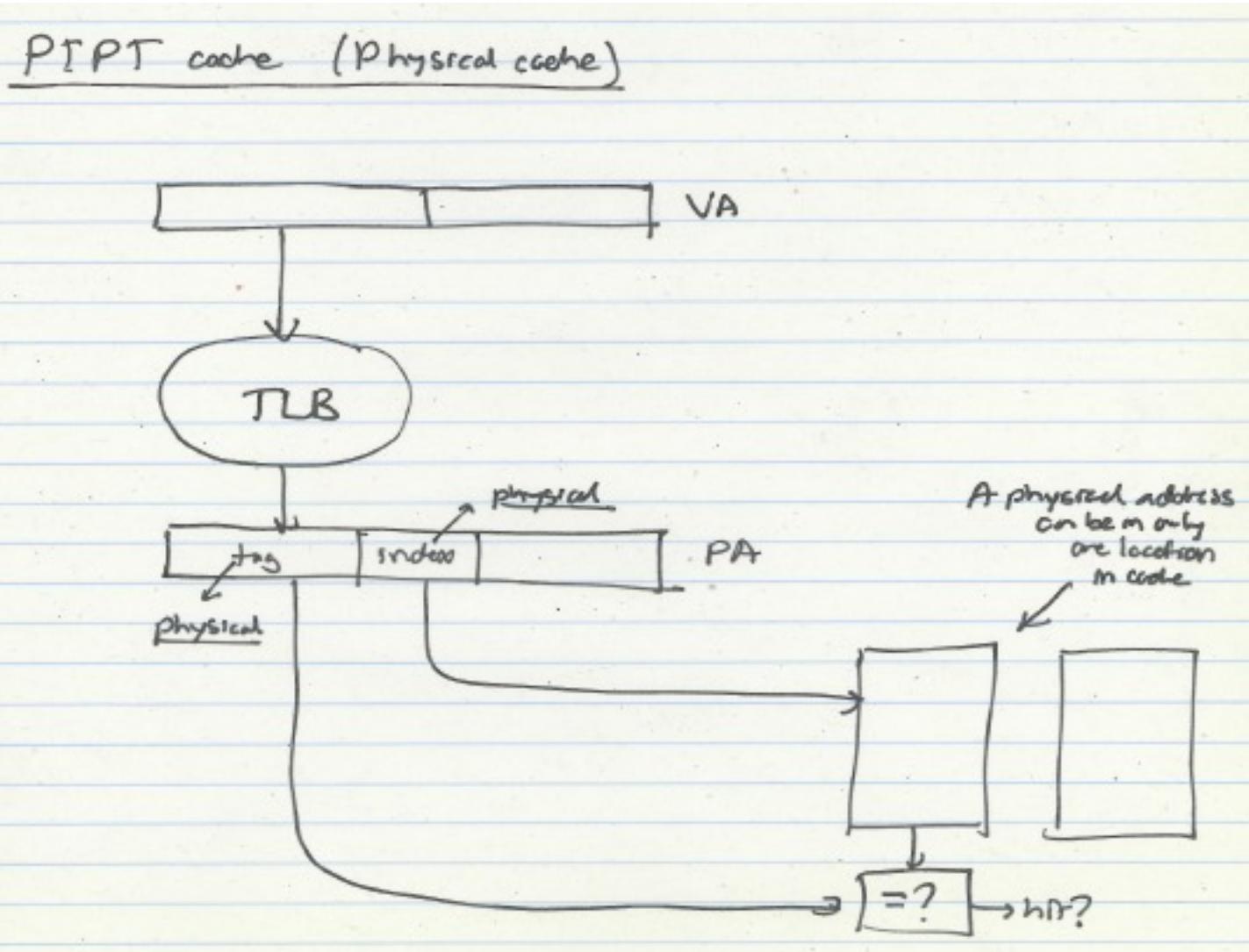
---

- Homonym: Same VA can map to two different PAs
  - Why?
    - VA is in different processes
- Synonym: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...
- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?

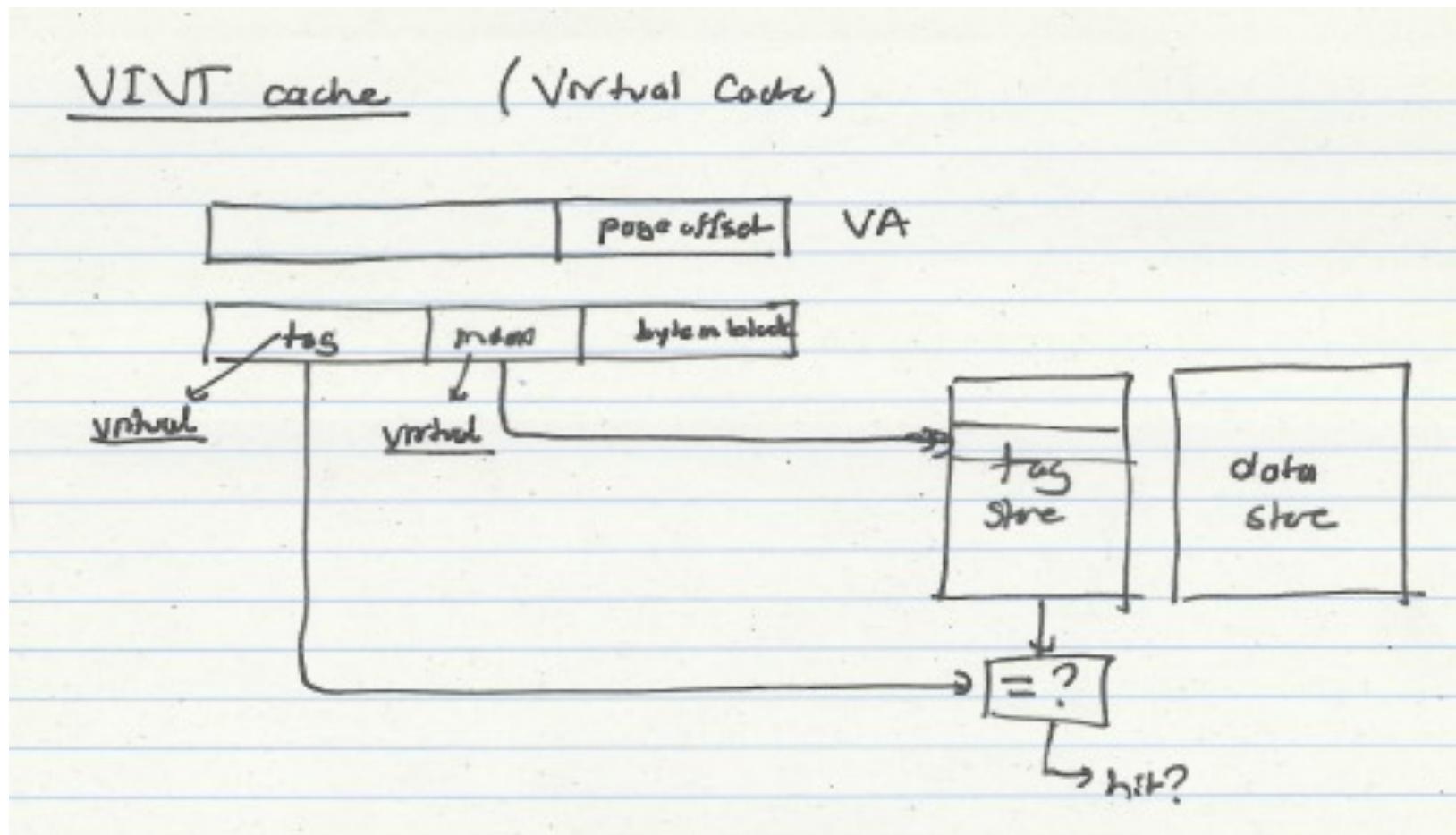
# Cache-VM Interaction



# Physical Cache

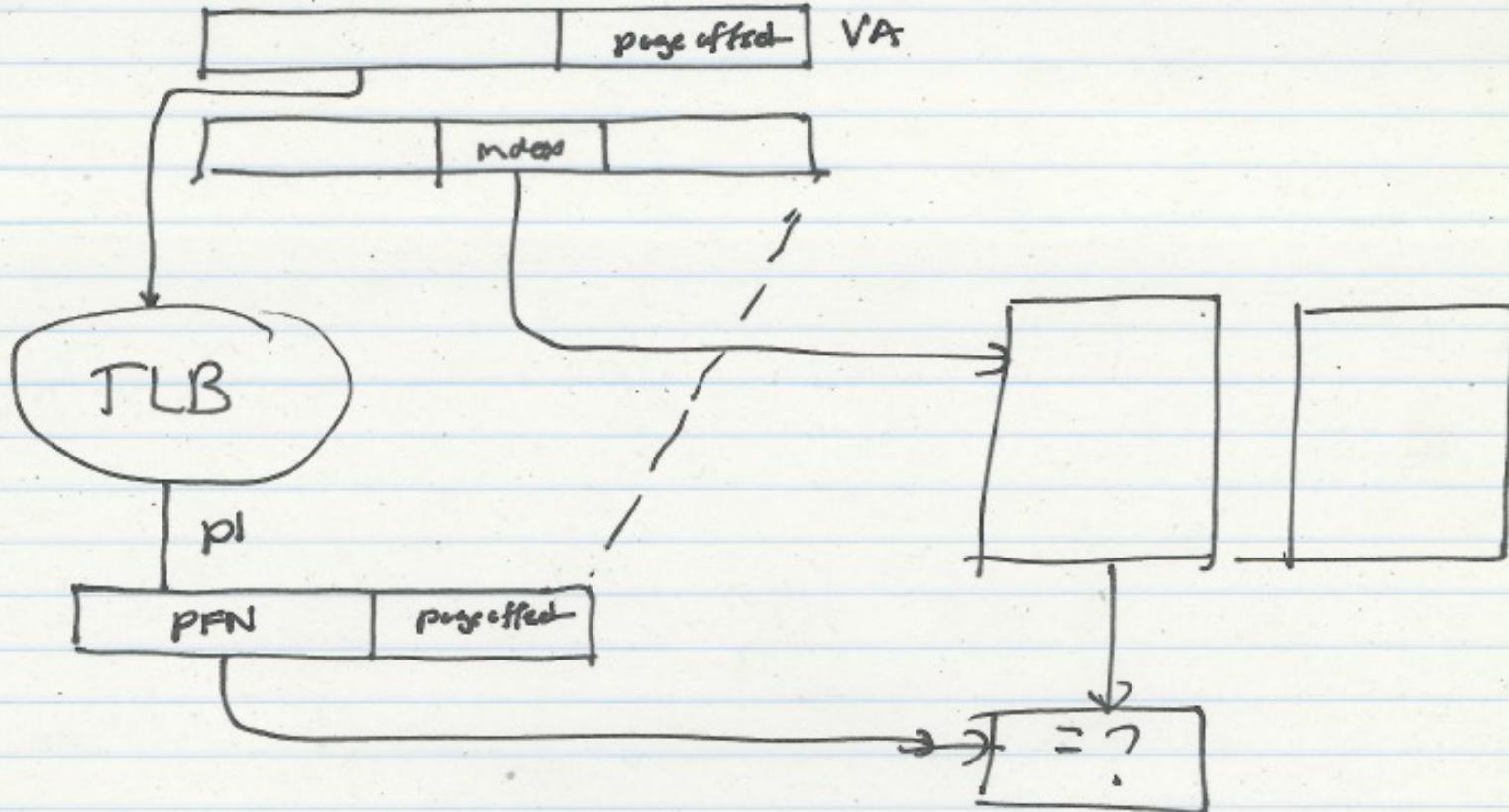


# Virtual Cache



# Virtual-Physical Cache

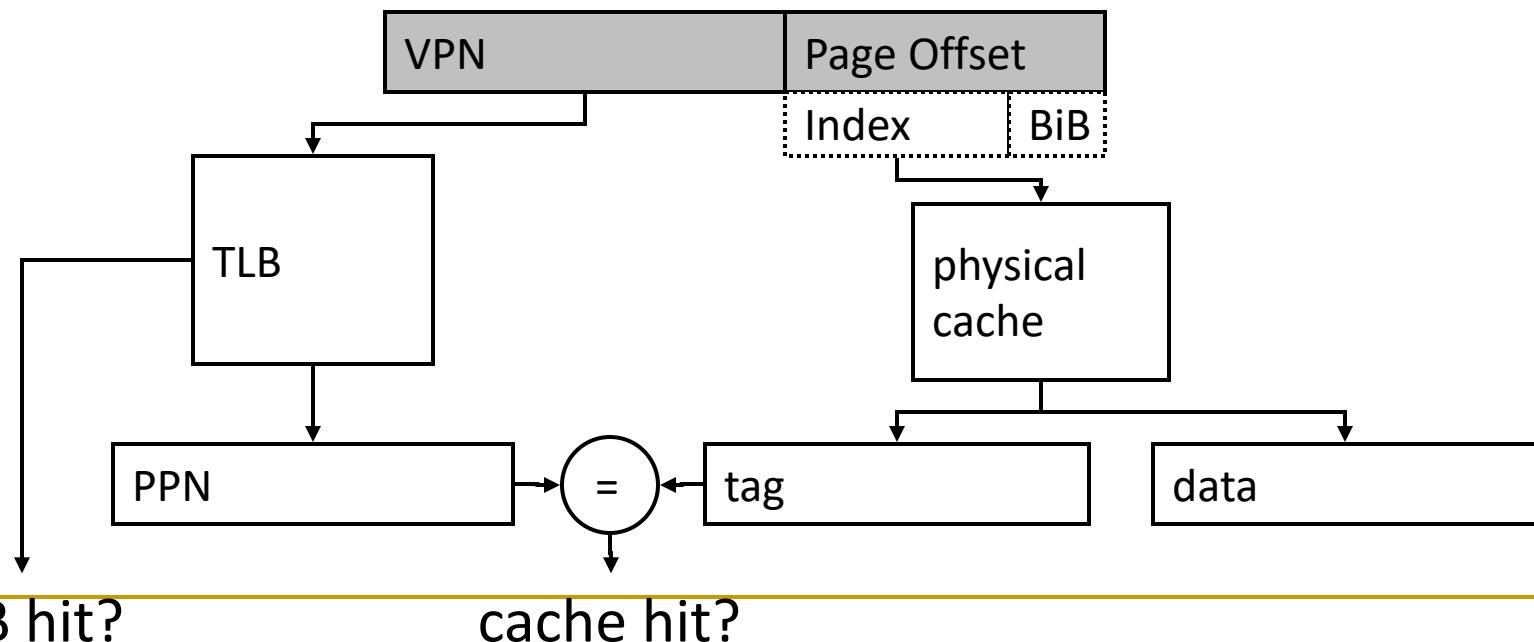
VIPT cache



Where can the same physical address be in the cache?

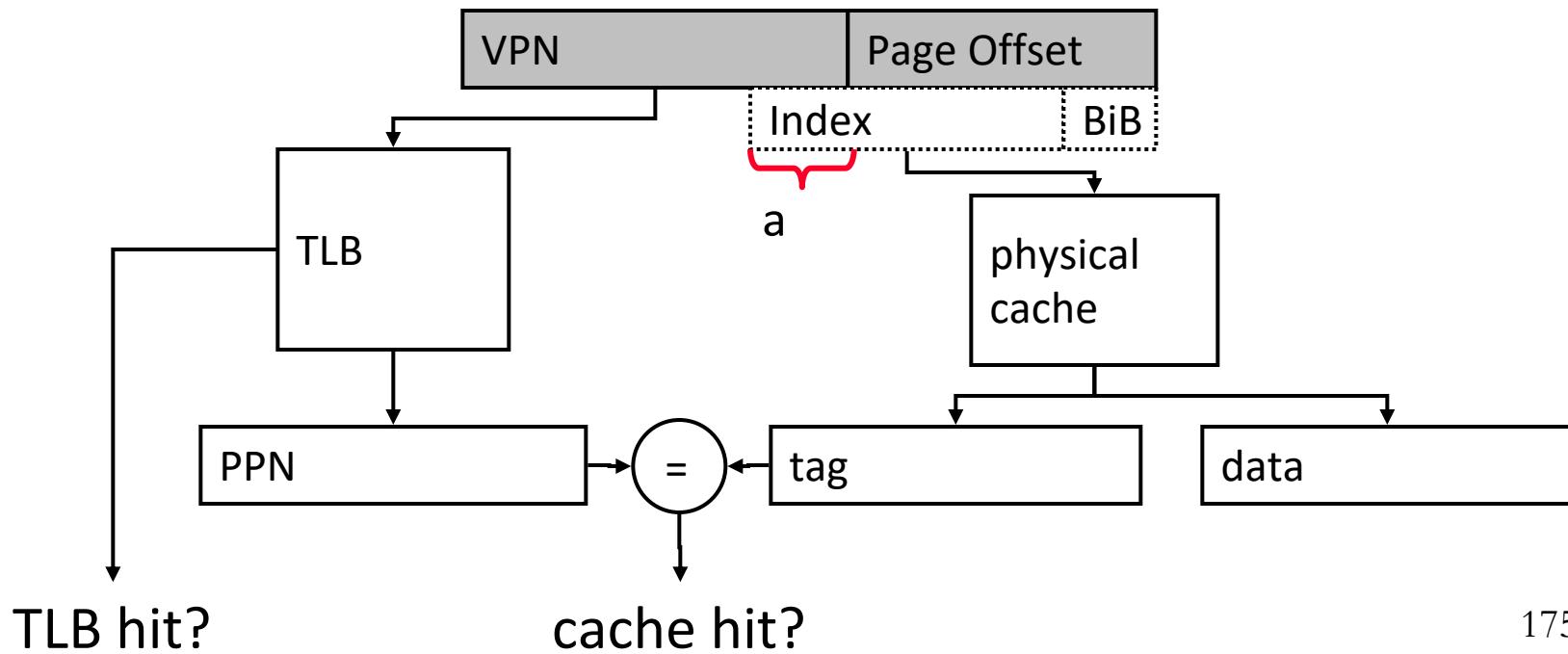
# Virtually-Indexed Physically-Tagged

- If  $C \leq (\text{page\_size} \times \text{associativity})$ , the cache index bits come only from page offset (same in VA and PA)
- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end



# Virtually-Indexed Physically-Tagged

- If  $C > (\text{page\_size} \times \text{associativity})$ , the cache index bits include VPN  
⇒ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?



# Some Solutions to the Synonym Problem

---

- Limit cache size to (page size times associativity)
  - get index from page offset
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K
- Restrict page placement in OS
  - make sure  $\text{index(VA)} = \text{index(PA)}$
  - Called page coloring
  - Used in many SPARC processors

# An Exercise (I)

---

We have a byte-addressable toy computer that has a physical address space of 512 bytes. The computer uses a simple, one-level virtual memory system. The page table is always in physical memory. The page size is specified as 8 bytes and the virtual address space is 2 KB.

## ***Part A.***

### **i. (1 point)**

How many bits of each virtual address is the virtual page number?

### **ii. (1 point)**

How many bits of each physical address is the physical frame number?

We would like to add a 128-byte *write-through* cache to enhance the performance of this computer. However, we would like the cache access and address translation to be performed simultaneously. In other words, we would like to index our cache using a virtual address, but do the tag comparison using the physical addresses (virtually-indexed physically-tagged). The cache we would like to add is direct-mapped, and has a block size of 2 bytes. The replacement policy is LRU. Answer the following questions:

**iii. (1 point)**

How many bits of a virtual address are used to determine which byte in a block is accessed?

**iv. (2 point)**

How many bits of a virtual address are used to index into the cache? Which bits exactly?

**v. (1 point)**

How many bits of the virtual page number are used to index into the cache?

**vi. (5 points)**

What is the size of the tag store in bits? Show your work.

**Part B.**

Suppose we have two processes sharing our toy computer. These processes share some portion of the physical memory. Some of the virtual page-physical frame mappings of each process are given below:

PROCESS 0	
Virtual Page	Physical Frame
Page 0	Frame 0
Page 3	Frame 7
Page 7	Frame 1
Page 15	Frame 3

PROCESS 1	
Virtual Page	Physical Frame
Page 0	Frame 4
Page 1	Frame 5
Page 7	Frame 3
Page 11	Frame 2

**vii. (2 points)**

Give a complete physical address whose data can exist in two different locations in the cache.

**viii. (3 points)**

Give the indexes of those two different locations in the cache.

# An Exercise (Concluded)

---

**ix. (5 points)**

We do not want the same physical address stored in two different locations in the 128-byte cache. We can prevent this by increasing the associativity of our virtually-indexed physically-tagged cache. What is the minimum associativity required?

**x. (4 points)**

Assume we would like to use a direct-mapped cache. Describe a solution that ensures that the same physical address is never stored in two different locations in the 128-byte cache.

# A Potpourri of Issues

# Trade-Offs in Page Size

---

## ■ Large page size (e.g., 1GB)

- Pro: Fewer PTEs required → Saves memory space
- Pro: Fewer TLB misses → Improves performance
- Con: Cannot have fine-grained permissions
- Con: Large transfers to/from disk
  - Even when only 1KB is needed, 1GB must be transferred
  - Waste of bandwidth/energy
  - Reduces performance
- Con: **Internal fragmentation**
  - Even when only 1KB is needed, 1GB must be allocated
  - Waste of space
  - Q: What is **external fragmentation?**

# Some System Software Tasks for VM

---

- Keeping track of which physical frames are free
- Allocating free physical frames to virtual pages
- Page replacement policy
  - When no physical frame is free, what should be removed?
- Sharing pages between processes
- Copy-on-write optimization
- Page-flip optimization