

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.  
Seoul National University

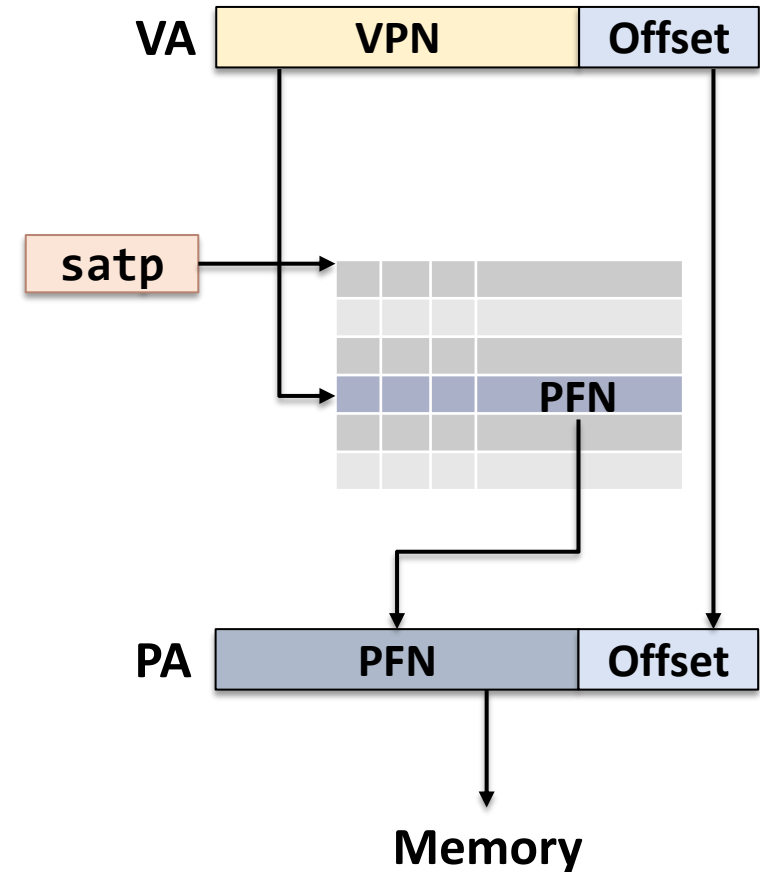
Fall 2025

# TLB



# Address Translation Steps

- For each memory reference,
  - Extract VPN from VA
  - Calculate the address of PTE
  - Read the PTE from memory
  - Extract PFN from PTE
  - Build PA
  - Read contents of PA from memory into register
- Which steps are expensive?



# The Problem

- Address translation is too slow
  - A simple linear page table doubles the cost of memory lookups
    - One for the page table, another to fetch the data
  - Multi-level page tables increase the cost further
- Goal: make address translation fast
  - Make fetching from a virtual address about as efficient as fetching from a physical address

# TLB

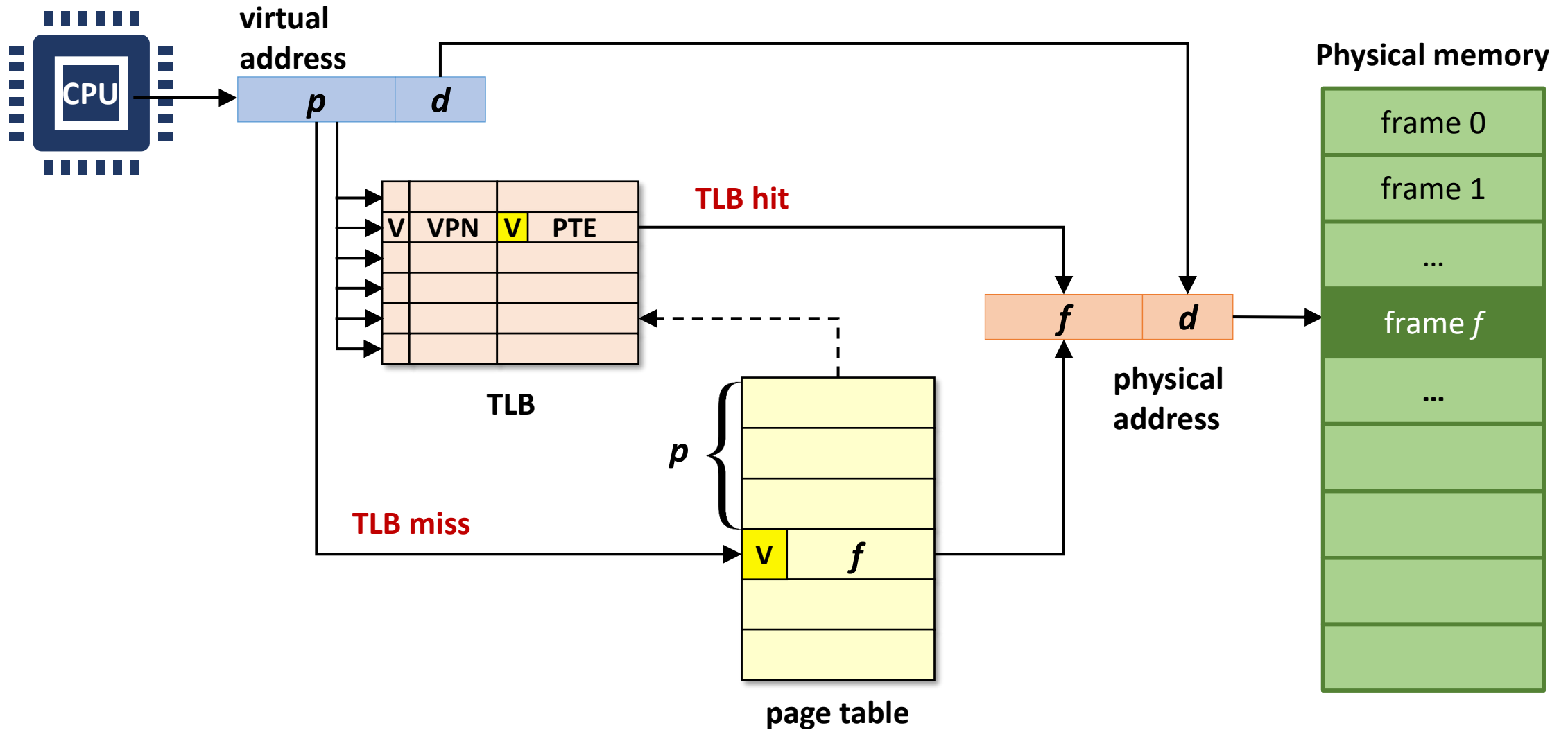
- Translation \_\_\_\_\_ Buffer
  - A hardware cache of popular virtual-to-physical address translations
  - Essential component which makes virtual memory possible
- TLB exploits locality
  - **Temporal locality**: an instruction or data item that has been recently accessed will likely be re-accessed soon
    - Instructions and data accesses in loops, ...
  - \_\_\_\_\_ **locality**: if a program accesses memory at address  $x$ , it will likely soon access memory near  $x$ 
    - Code execution, array traversal, stack accesses, ...

# TLB Organization

- TLB is implemented in hardware
  - Processes only use a handful of pages at a time
    - 16~256 entries in TLB is typical
  - Usually fully associative
    - All entries looked up in parallel
    - But may be set associative to reduce latency
  - Replacement policy: LRU (Least Recently Used)
  - TLB actually caches the whole PTEs, not just PFNs

| Valid | Tag (VPN) | Value (PTE) |   |   |      |     |        |
|-------|-----------|-------------|---|---|------|-----|--------|
| 1     | 0x1000    | V           | R | M | Prot | PFN | 0x1234 |
| 1     | 0x2400    | V           | R | M | Prot | PFN | 0x8800 |
| 0     | -         | -           |   |   |      |     |        |

# Address Translation with TLB



# Handling TLB Misses

## ■ Software-managed TLB

- CPU traps into OS upon TLB miss
- OS finds right PTE and loads it into TLB
- CPU ISA has (privileged) instructions for TLB manipulation
- Page tables can be in any format convenient for OS (flexible)

## ■ Hardware-managed TLB

- CPU knows where page tables are in memory
  - e.g., CR3 (or PDBR) register in IA-32 / Intel 64, satp in RISC-V
- OS maintains page tables
- CPU “walks” the page table and fills TLB
- Page tables have to be in hardware-defined format

# TLB on Context Switches

- Flush TLB on each context switch
  - TLB is flushed automatically when CR3 is changed in a hardware-managed TLB
  - Some architectures support the pinning of pages into TLB
    - For pages that are globally shared among processes (e.g., kernel pages)
    - MIPS, Intel, etc.
- Track which entries are for which process
  - Tag each TLB entry with an ASID (Address Space ID)
  - A privileged register holds the ASID of the current process
  - MIPS supports 8-bit ASID
    - Why not use PID?
    - What if there are more than 256 processes running?
  - RISC-V supports up to 16-bit ASID for Sv39/Sv48 (stored in satp register)

# TLB on Multi-core

## ■ TLB coherence

- Page-table changes may leave stale entries in the TLBs
- Flushing the local TLB is not enough
- Unlike memory caches, TLBs of different cores are not maintained coherent by hardware
- TLB coherence should be restored by the OS

## ■ TLB

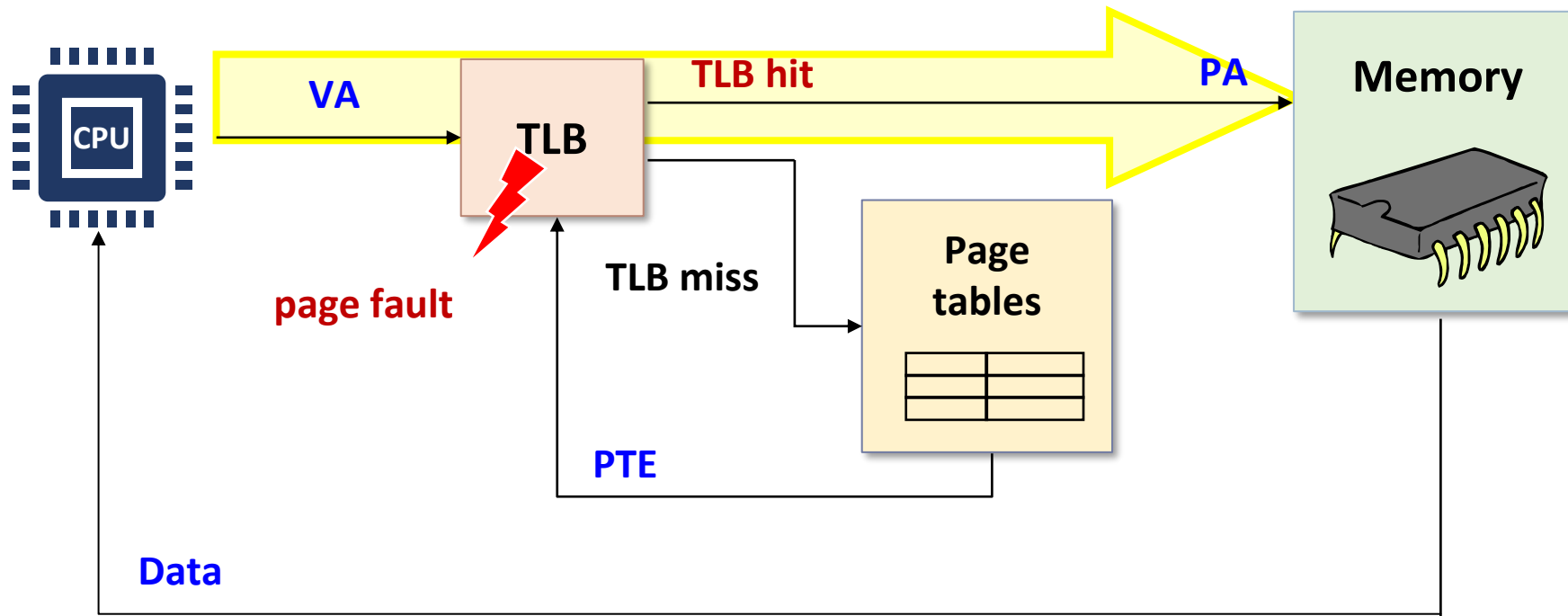
- The initiating core sends an IPI (Inter-Processor Interrupt) to the remote cores
- The remote cores invalidate their TLBs (may need to flush the entire TLB)
- The IPI may take several hundreds of cycles

# TLB Performance

- TLB is the source of many performance problems
  - Performance metric: hit rate, lookup latency, ...
- Increase TLB \_\_\_\_\_ (= #TLB entries \* Page size)
  - Use **superpages** (or **hugepages**): e.g., 2MB, 1GB page support in Intel 64
  - Increase the TLB size
- Use multi-level TLBs
  - e.g., Intel Haswell (4KB pages): L1 ITLB 128 entries (4-way), L1 DTLB 64-entries (4-way), L2 STLB 1024 entries (8-way)
- Change your algorithms and data structures to be TLB-friendly

# From CPU to Memory

- A process is executing on the CPU, and it issues a read to a virtual address



# Load Example

- The common case

- The load instruction goes to the TLB in the MMU
- TLB does a lookup using the virtual page number of the address
- The page number matches, returning a PTE
- TLB checks the valid / protection bits in the PTE
- TLB validates that the PTE protection allows loads
- PTE specifies which physical frame holds the page
- MMU combines the physical frame and offset into a physical address
- MMU then reads from that physical address, returns value to CPU

# On TLB Miss

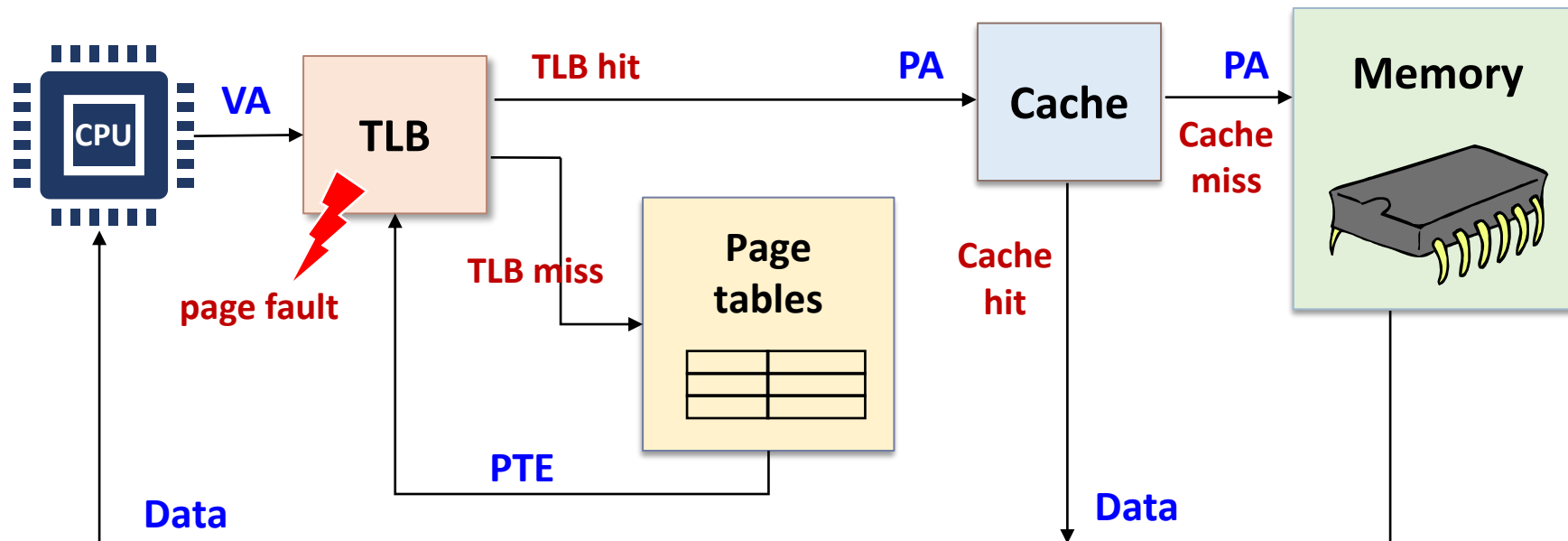
- **Hardware-managed TLB**
  - MMU loads PTE from page table in memory
  - OS has already set up the page tables so that the hardware can access it directly
  - OS is not involved in this step
- **Software-managed TLB**
  - Trap to the OS
  - OS does lookup in page tables, loads PTE into TLB
  - OS returns from exception
- At this point, there is a valid PTE for the address in the TLB
- TLB restarts translation

# On Page Faults

- PTE can indicate a page fault
  - Read/Write/Execute – operation not permitted on page
  - Invalid – virtual page not allocated or page not in physical memory
- TLB traps to the OS
  - Read/Write/Execute – OS usually will send fault back to the process, or might be playing tricks (e.g., copy on write, mapped files)
  - Invalid (Not allocated) – OS sends fault to the process (e.g., segmentation fault)
  - Invalid (Not in physical memory) – OS allocates a frame, reads from disk, and maps PTE to physical frame.

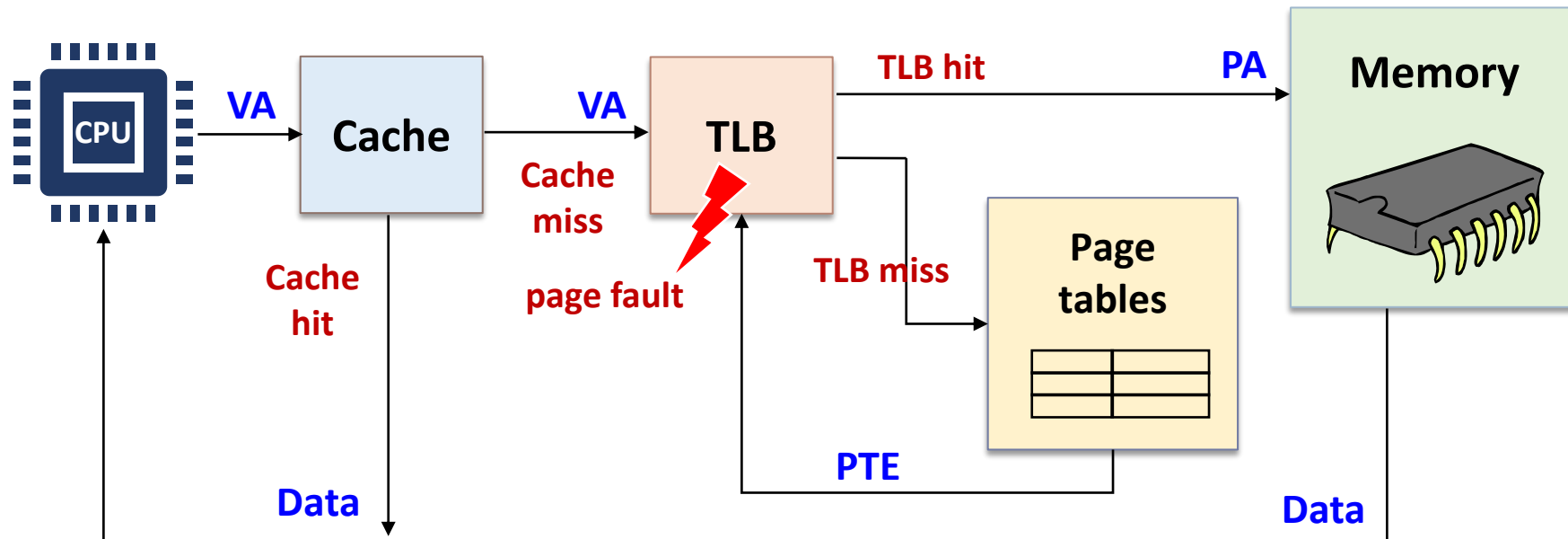
# Integrating VM and Cache (I)

- Physically indexed, physically tagged cache
  - Allows multiple processes to have blocks in cache
  - Allows multiple processes to share pages
  - Address translation is on the critical path



# Integrating VM and Cache (2)

- Virtually indexed, virtually tagged cache
  - \_\_\_\_\_ problem
    - Each process has a different translation of the same virtual address
  - Address \_\_\_\_\_ or aliases problem
    - Two different virtual addresses point to the same physical address



# Integrating VM and Cache (3)

- Virtually indexed, physically tagged cache
  - Use virtual address to access the TLB and cache in parallel
  - TLB produces the PFN – which must match the physical tag of the accessed cache line for it to be a “hit”

|    |     |             |        |
|----|-----|-------------|--------|
| VA | VPN | page offset |        |
| PA | PFN | page offset |        |
| PA | tag | set         | offset |

