CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 9 – Virtual Memory

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 8

- Protection and translation required for multiprogramming
  - Base and bounds was early simple scheme
- Page-based translation and protection avoids need for memory compaction, easy allocation by OS
  - But need to indirect in large page table on every access
- Address spaces accessed sparsely
  - Can use multi-level page table to hold translation/protection information, but implies multiple memory accesses per reference
- Address space access with locality
  - Can use “translation lookaside buffer” (TLB) to cache address translations (sometimes known as address translation cache)
  - Still have to walk page tables on TLB miss, can be hardware or software talk
- Virtual memory uses DRAM as a “cache” of disk memory, allows very cheap main memory

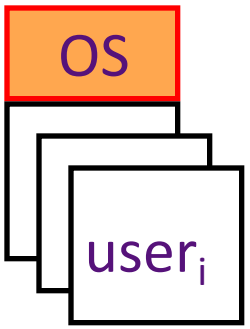
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

several users, each with their private address space and one or more shared address spaces

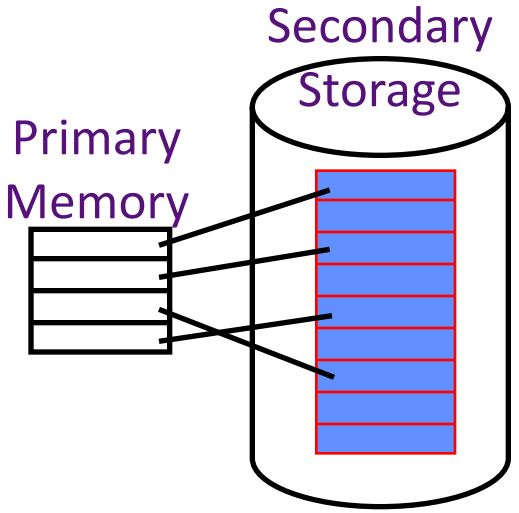
page table ⇔ name space



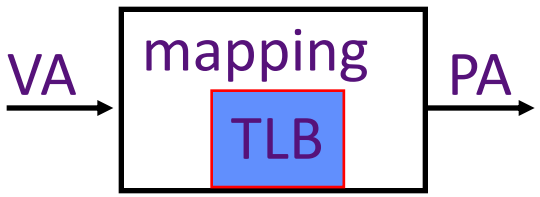
Demand Paging

Provides the ability to run programs larger than the primary memory

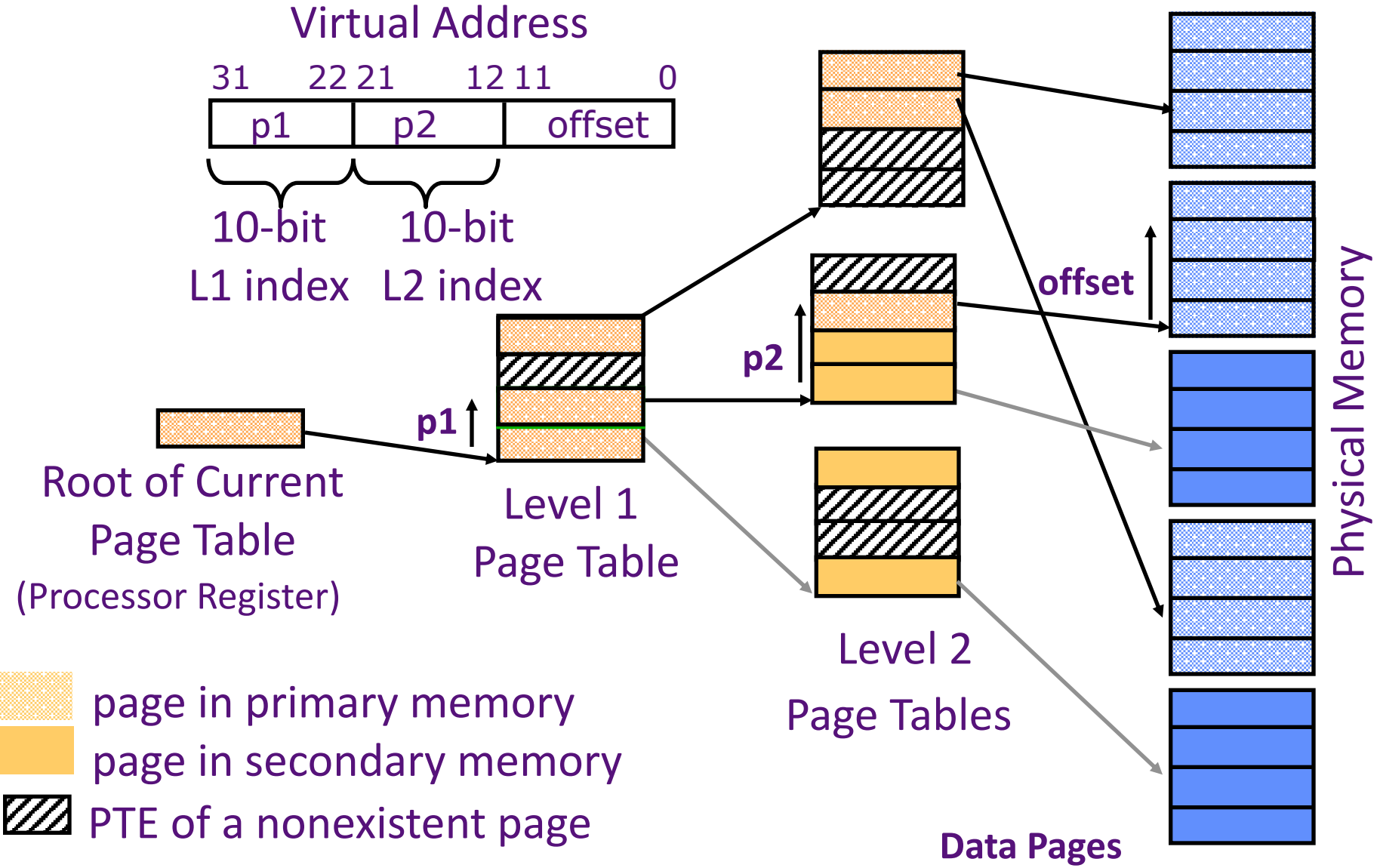
Hides differences in machine configurations



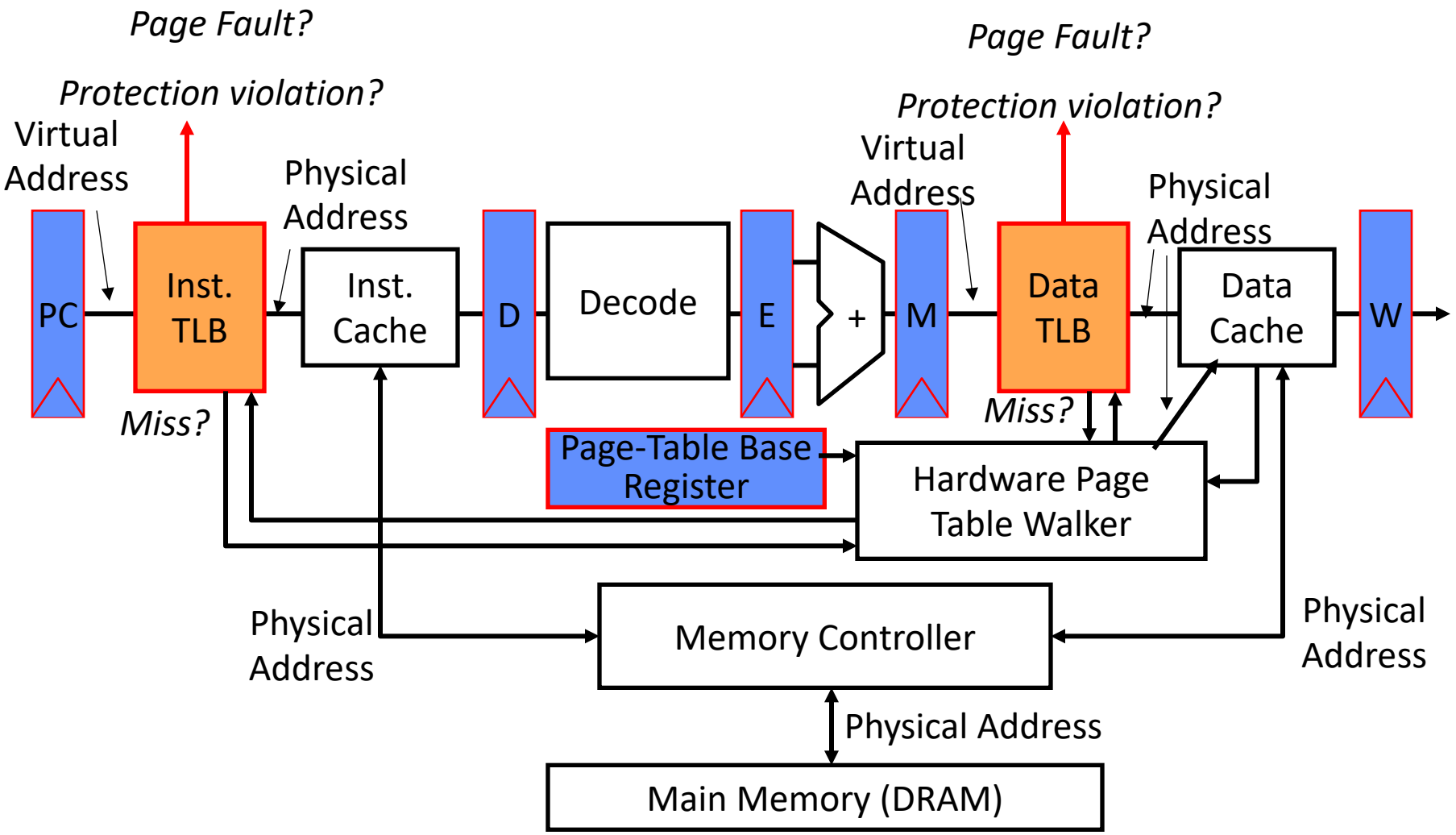
The price is address translation on each memory reference



Recap: Hierarchical Page Table

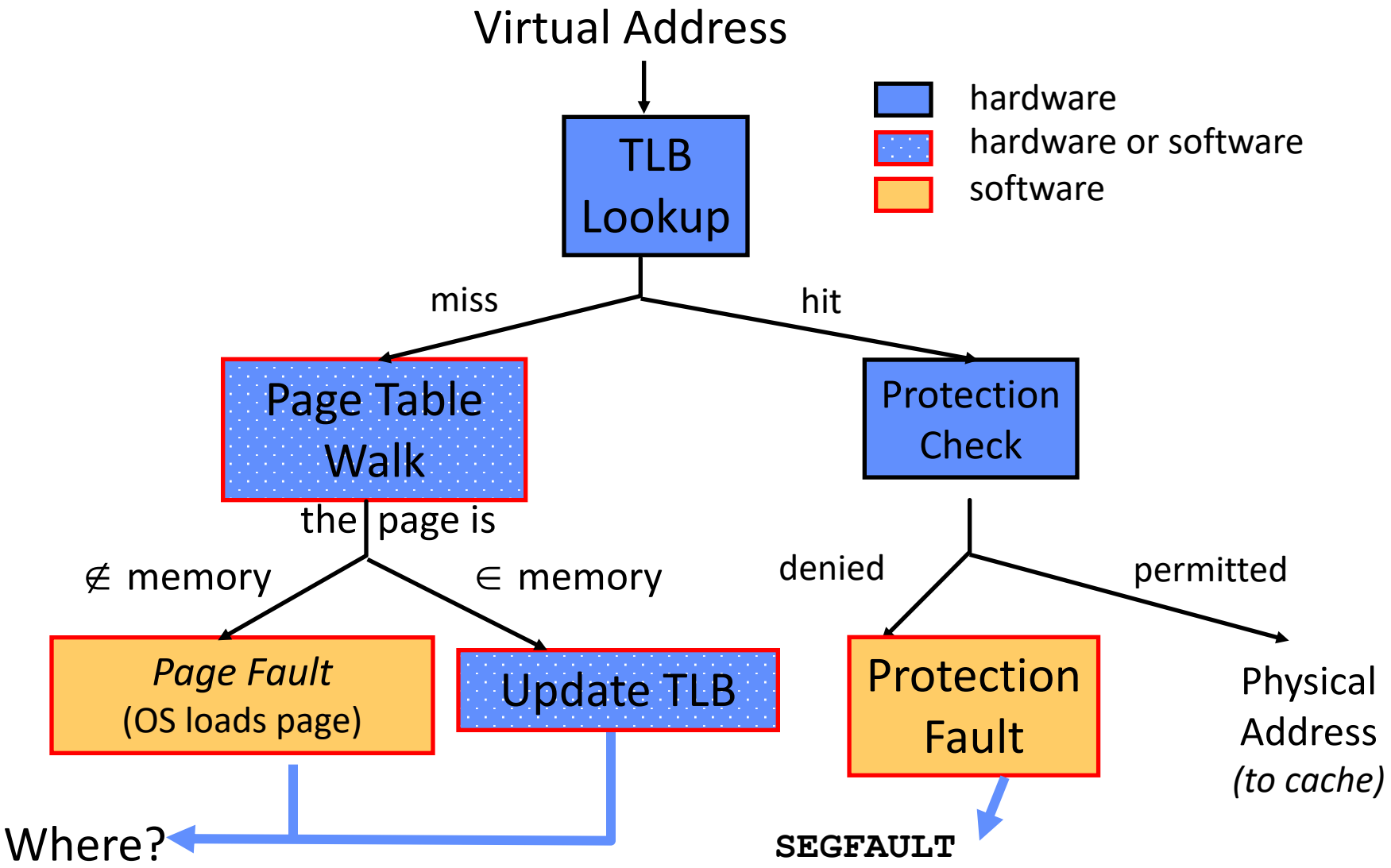


Recap: Page-Based Virtual-Memory Machine
(Hardware Page-Table Walk)



- Assumes page tables held in untranslated physical memory

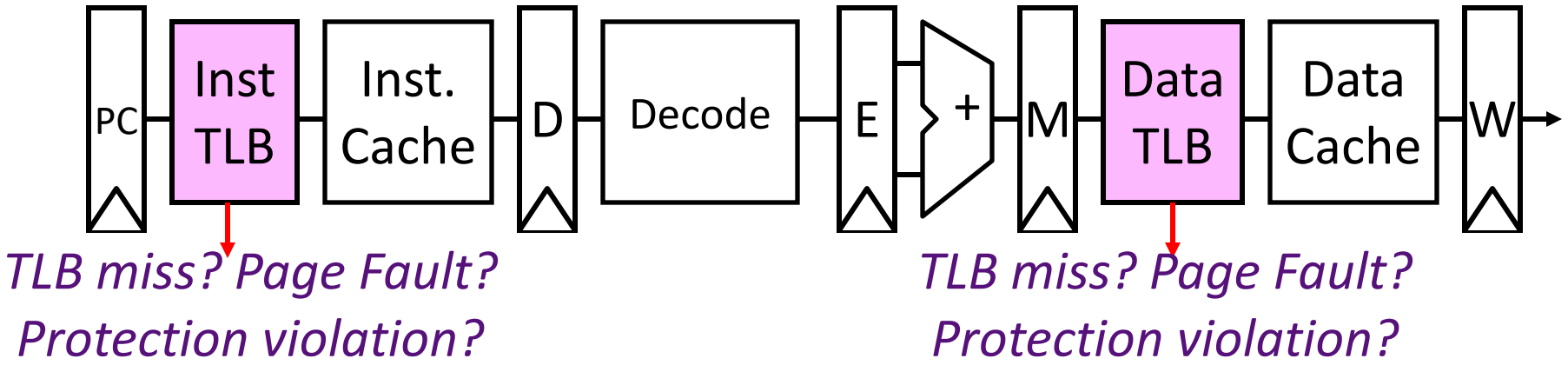
Address Translation:
putting it all together



Page Fault Handler

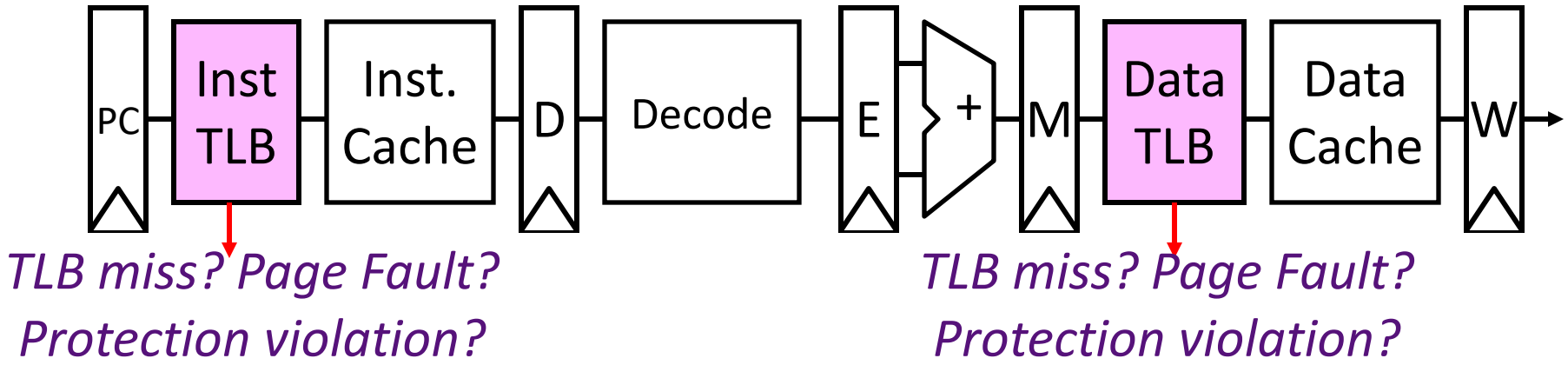
- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    - Pseudo-LRU replacement policy, implemented in software
- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by the OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

Handling VM-related exceptions



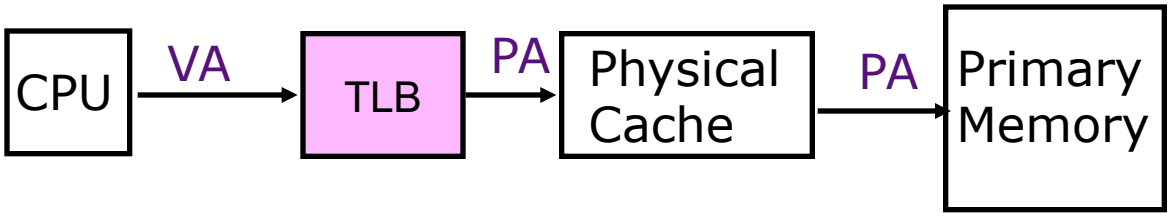
- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Handling page fault (e.g., page is on disk) needs *restartable* exception so software handler can resume after retrieving page
  - Precise exceptions are easy to restart
  - Can be imprecise but restartable, but this complicates OS software
- A protection violation may abort process
  - But often handled the same as a page fault

Address Translation in CPU Pipeline

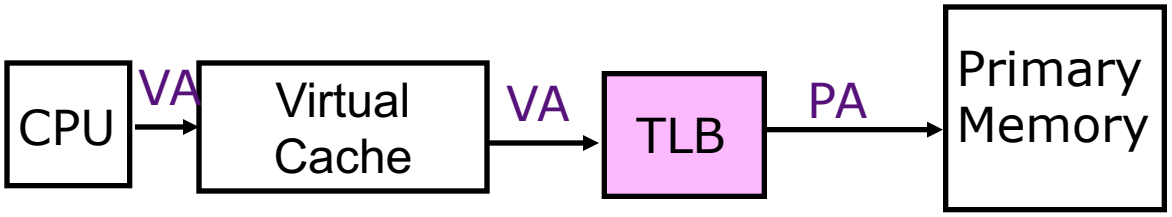


- Need to cope with additional latency of TLB:
  - slow down the clock?
  - pipeline the TLB and cache access?
  - virtual address caches
  - parallel TLB/cache access

Virtual-Address Caches



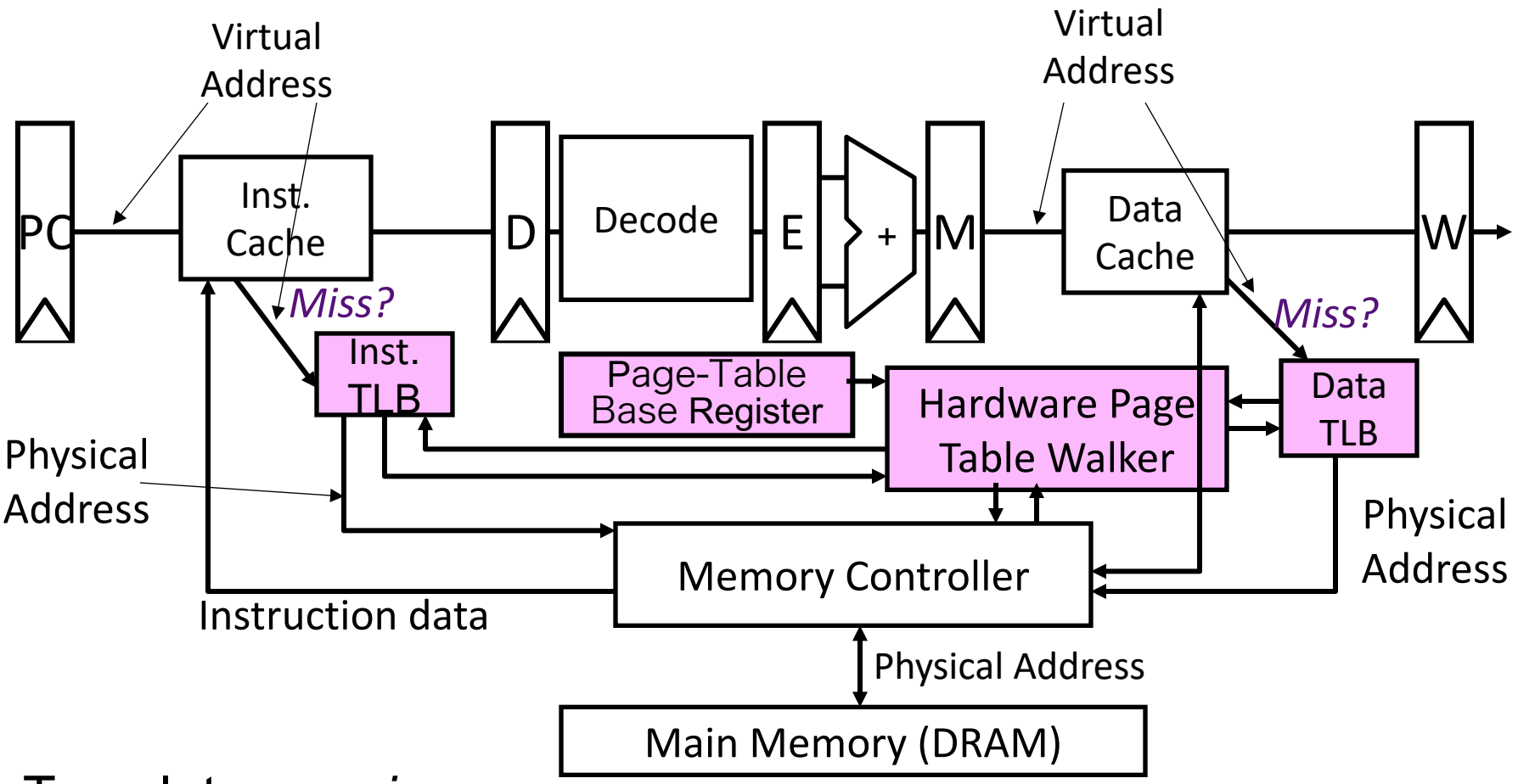
Alternative: place the cache before the TLB



(StrongARM)

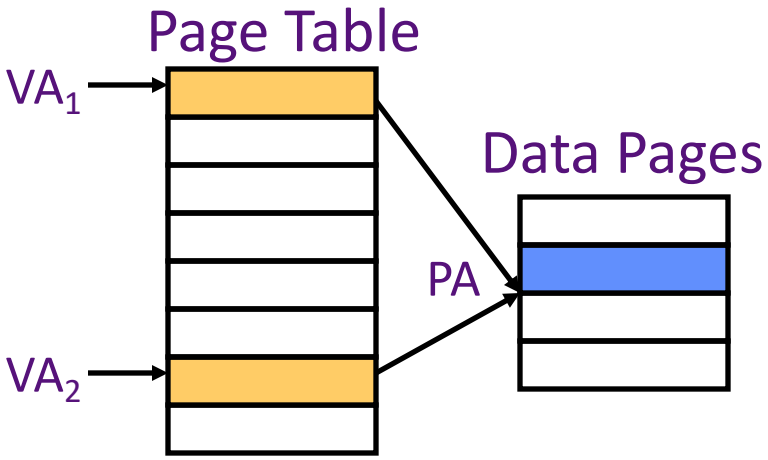
- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)
- maintaining cache coherence (-)

Virtually Addressed Cache (Virtual Index/Virtual Tag)



Translate on *miss*

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

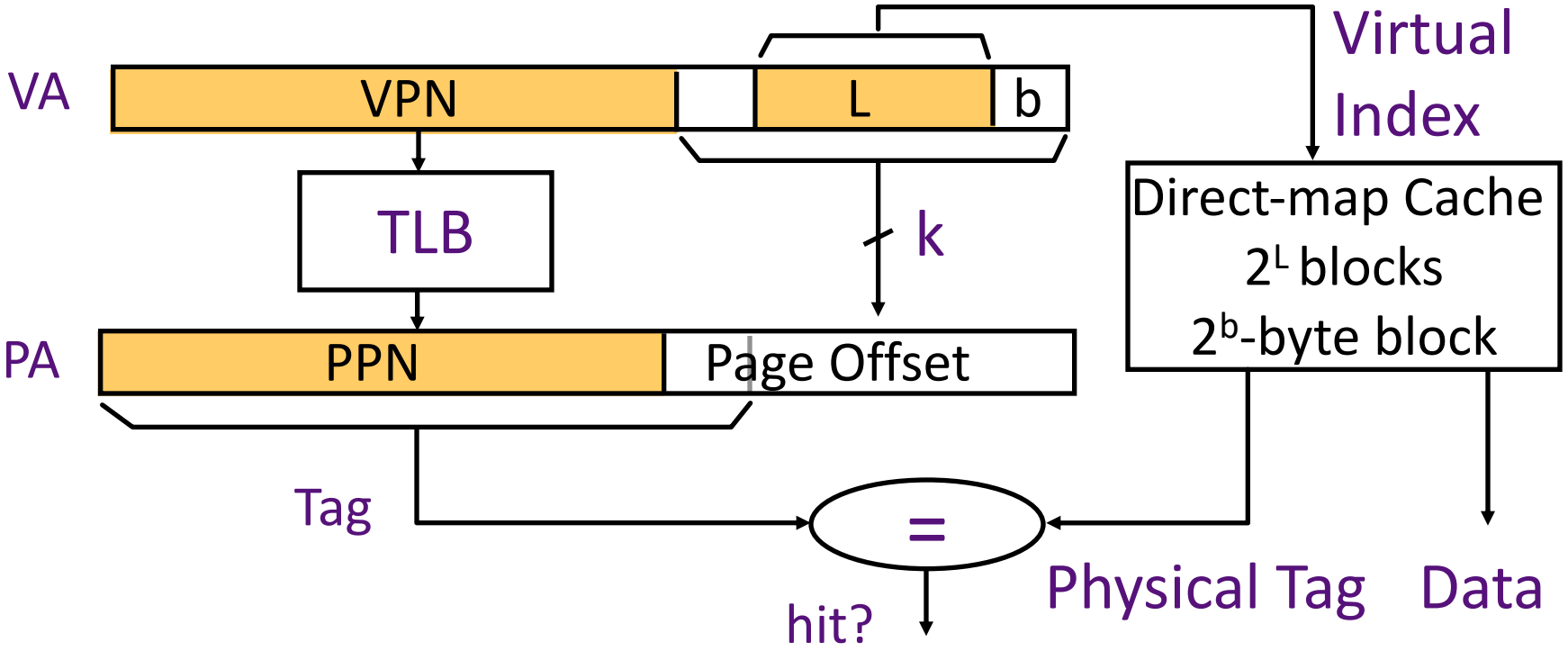
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Prevent aliases coexisting in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)

Concurrent Access to TLB & Cache (Virtual Index/Physical Tag)



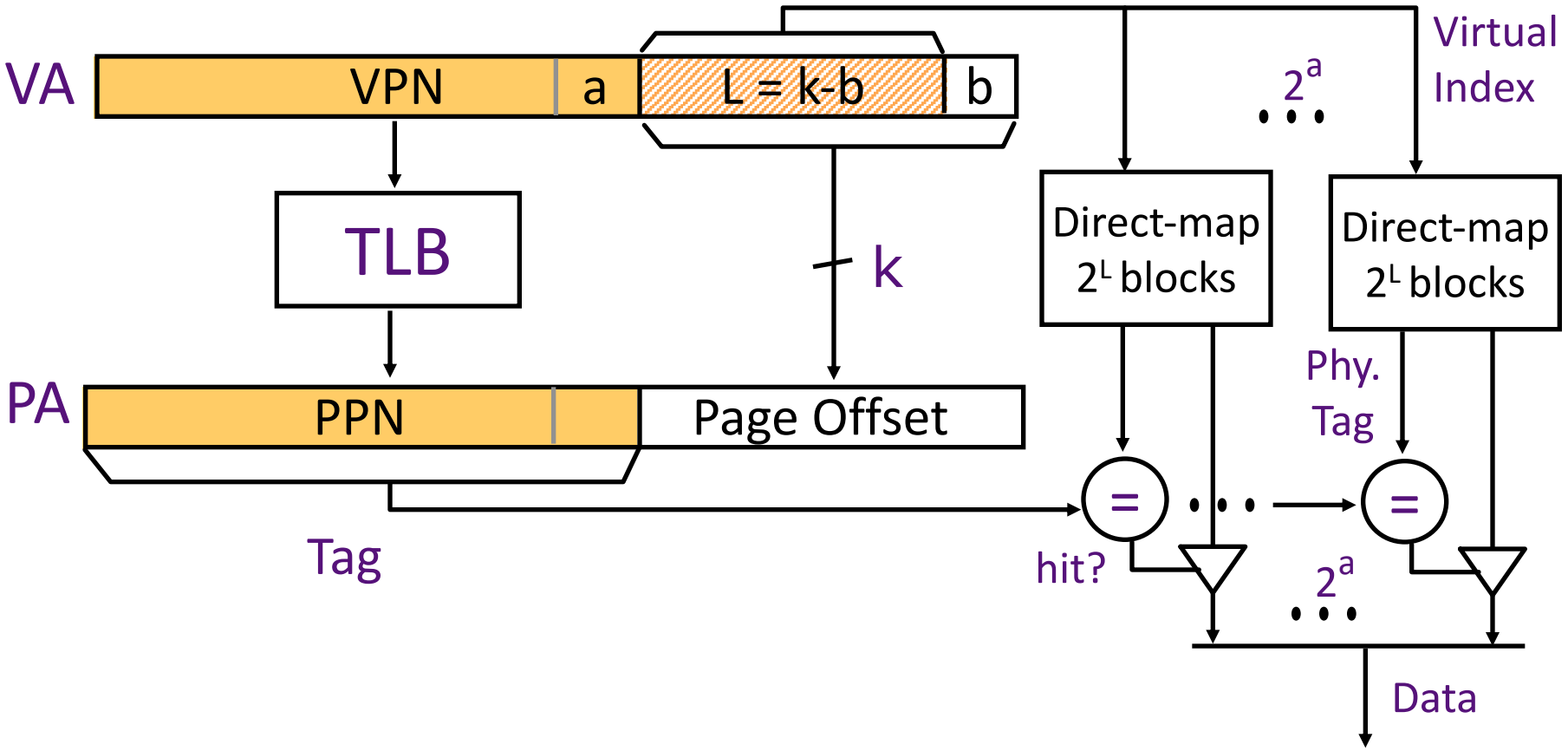
Index L is available without consulting the TLB

→ *cache and TLB accesses can begin simultaneously!*

Tag comparison is made after both accesses are completed

Cases: $L + b = k$, $L + b < k$, $L + b > k$

Virtual-Index Physical-Tag Caches:
Associative Organization



After the PPN is known, 2^a physical tags are compared

How does this scheme scale to larger caches?

CS152 Administrivia

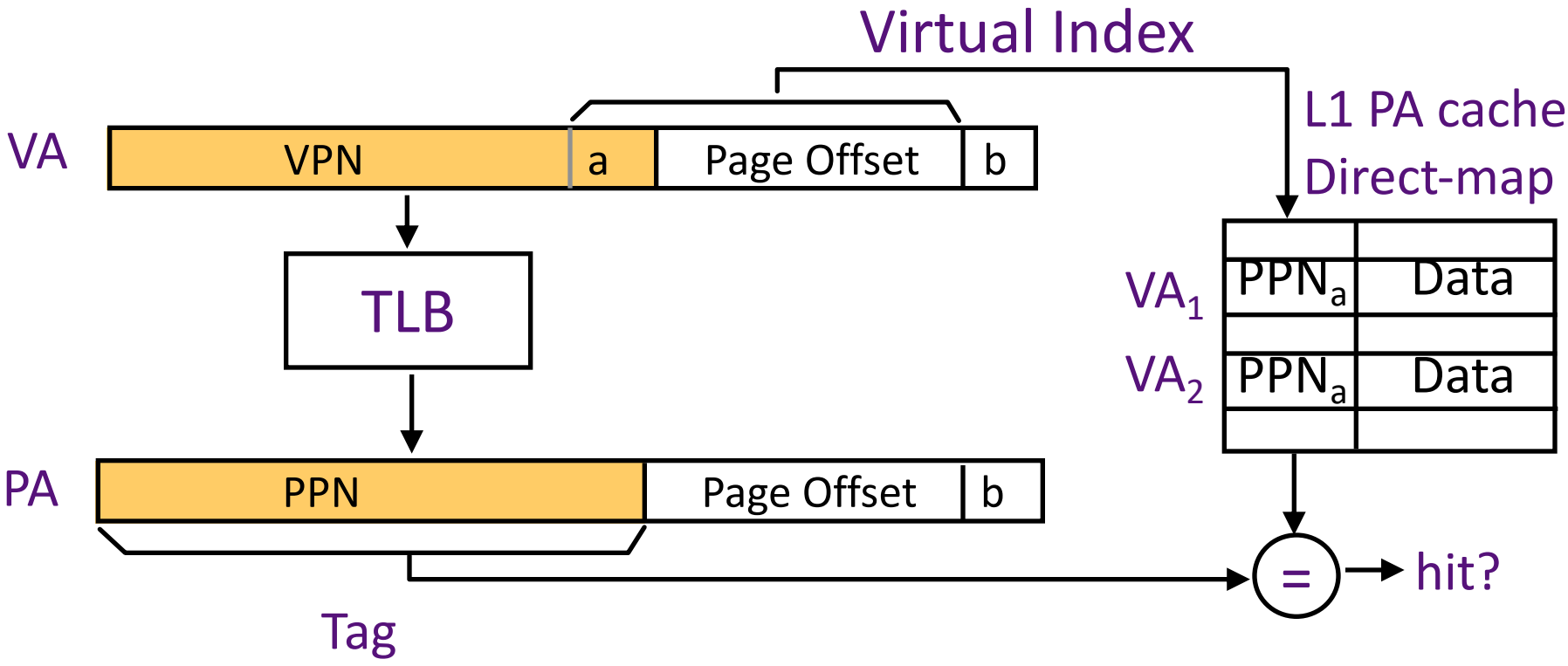
- PS 2 due Wednesday Feb 27
- Midterm in class Monday March 6
  - Covers lectures 1 – 9, plus assigned problem sets, labs, book readings
- Lab 2 due Monday March 11

CS252 Administrivia

- Start thinking of class projects and forming teams of two
- Proposal due Wednesday February 27th
- Proposal should be one page PDF including:
  - Title
  - Team member names
  - What are you trying to do?
  - How is it done today?
  - What is your idea for improvement and why do you think you'll be successful
  - What infrastructure are you going to use for your project?
  - Project timeline with milestones
- Mail PDF of proposal to instructors
- Give a <5-minute presentation in class in discussion section time on March 11th
- No discussion on Mnoday March 4th – midterm!

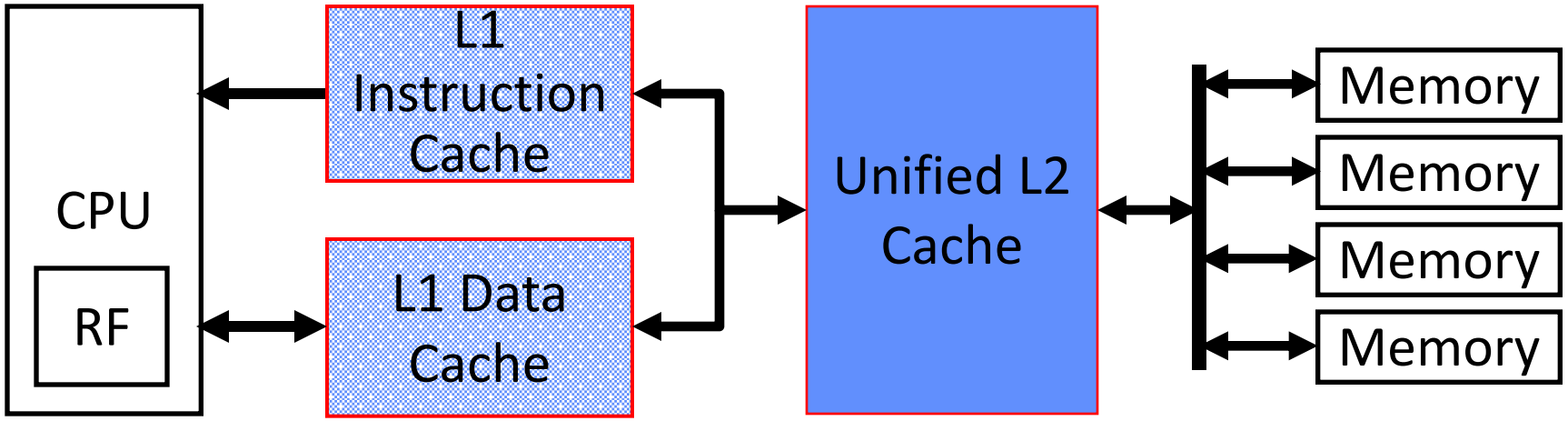
Concurrent Access to TLB & Large L1

The problem with L1 > Page size



Can VA_1 *and* VA_2 *both map to* PA *?*

A solution via Second-Level Cache

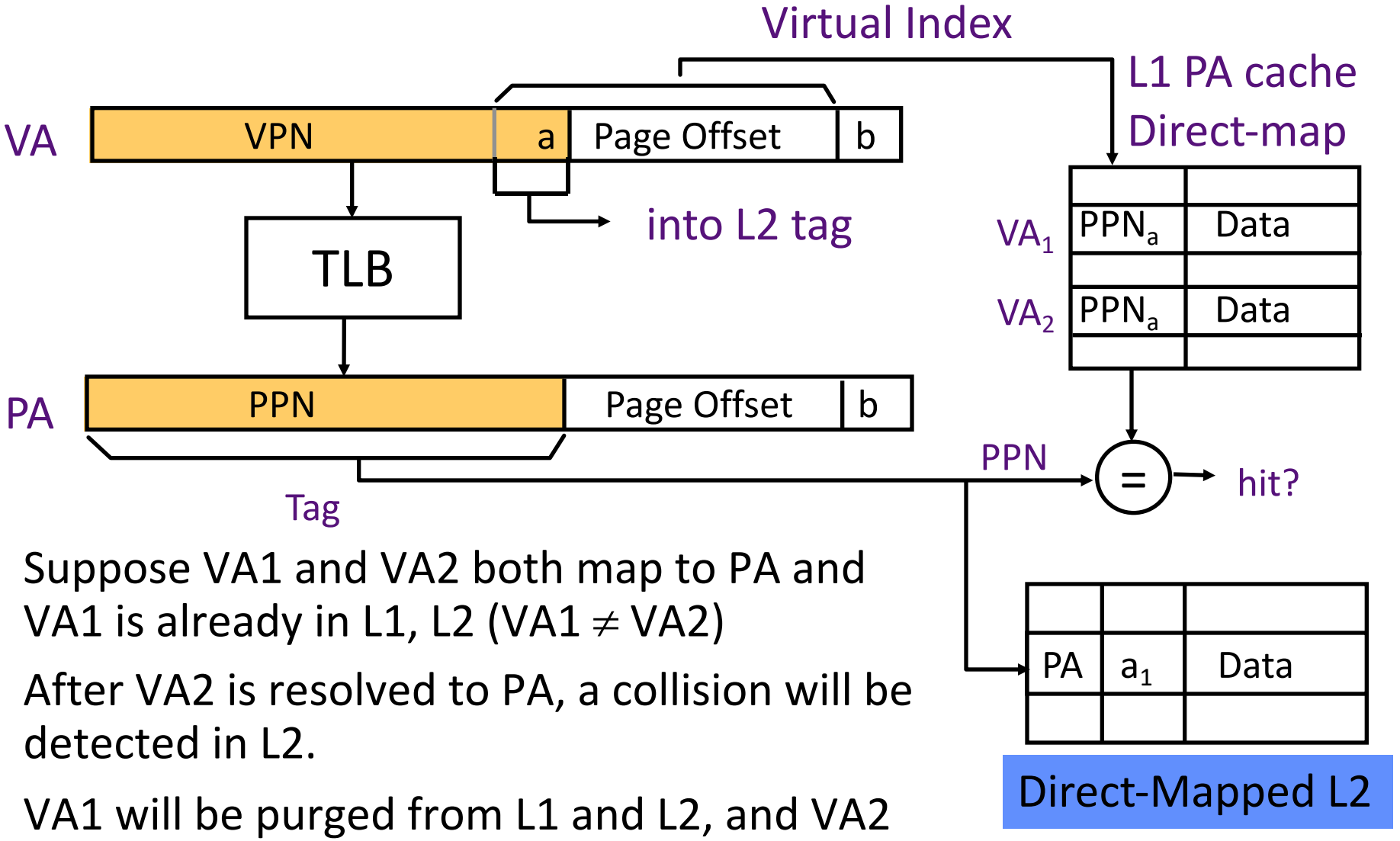


Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is “inclusive” of both Instruction and Data caches

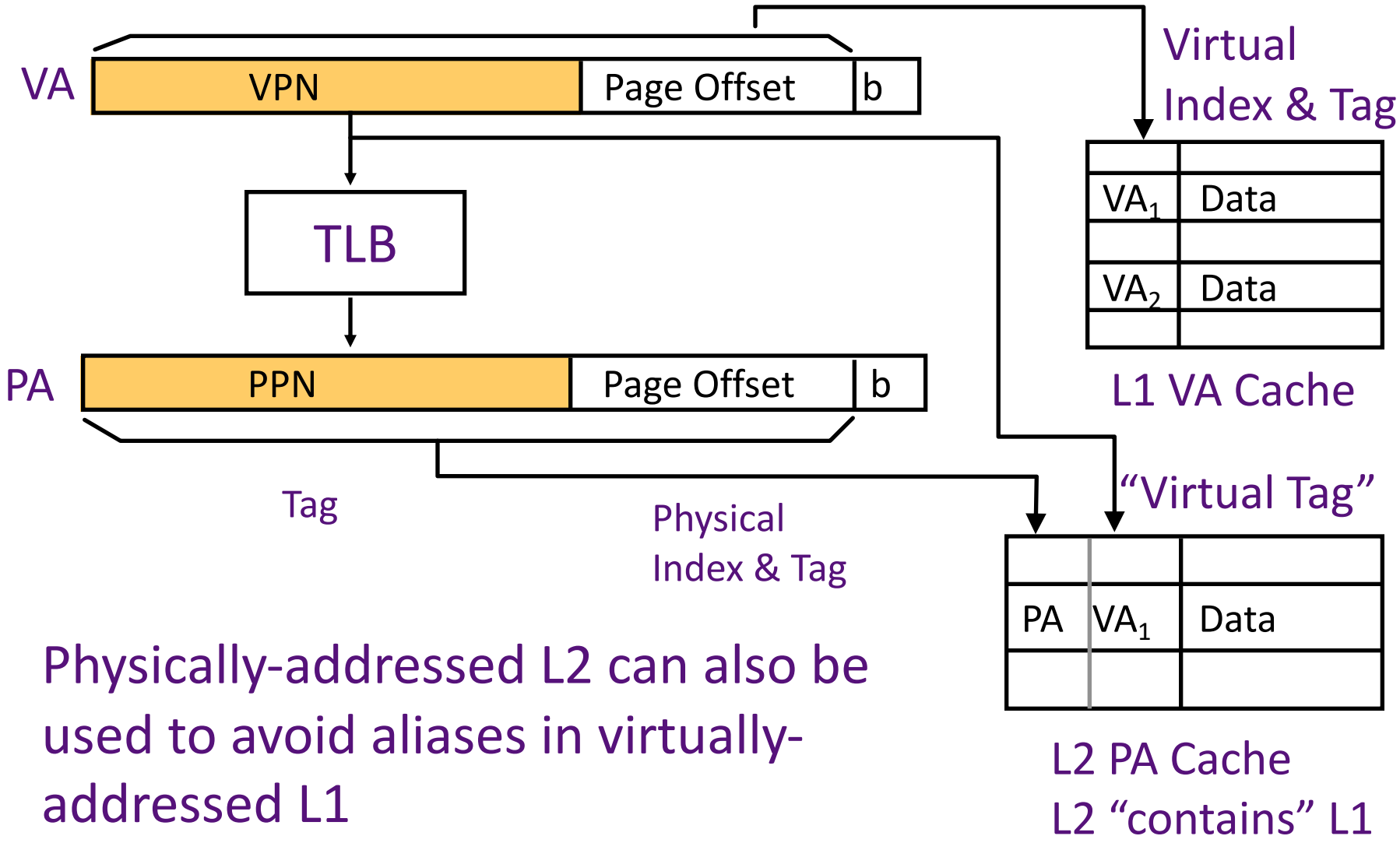
- Inclusive means L2 has copy of any line in either L1

Anti-Aliasing Using L2 [*MIPS R10000,1996*]



- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, a collision will be detected in L2.
- VA1 will be purged from L1 and L2, and VA2 will be loaded ⇒ *no aliasing !*

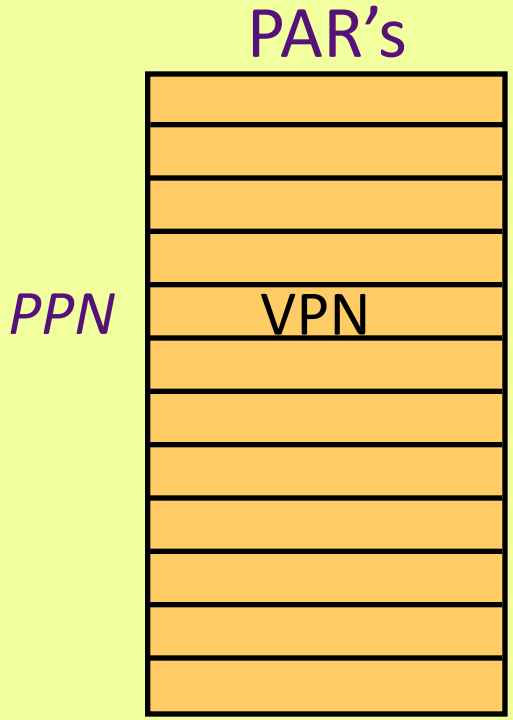
Anti-Aliasing using L2 for a Virtually Addressed L1



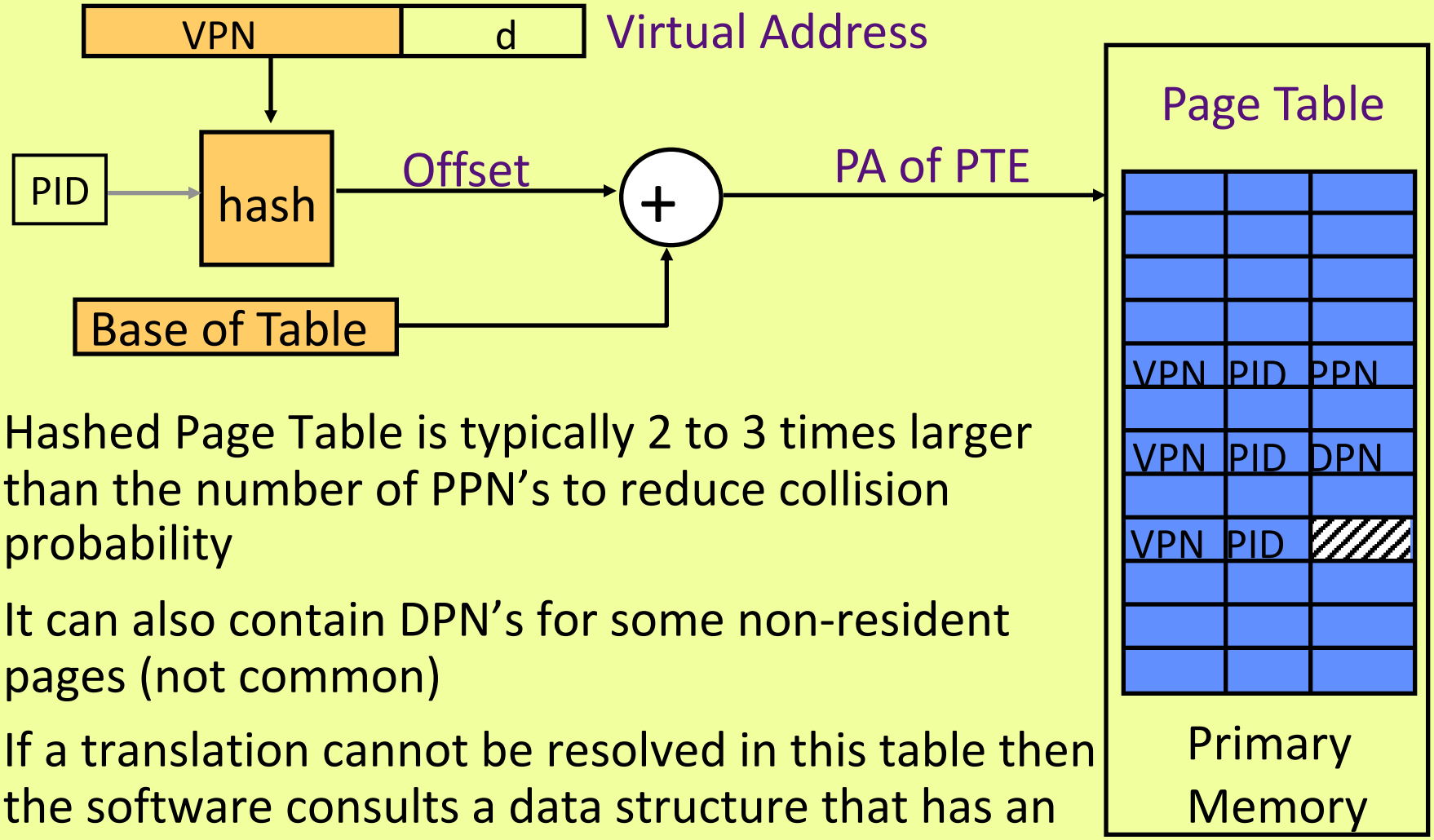
Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1

Atlas Revisited

- One PAR for each physical page
- PAR's contain the VPN's of the pages *resident in primary memory*
- *Advantage:* The size is proportional to the size of the primary memory
- *What is the disadvantage ?*

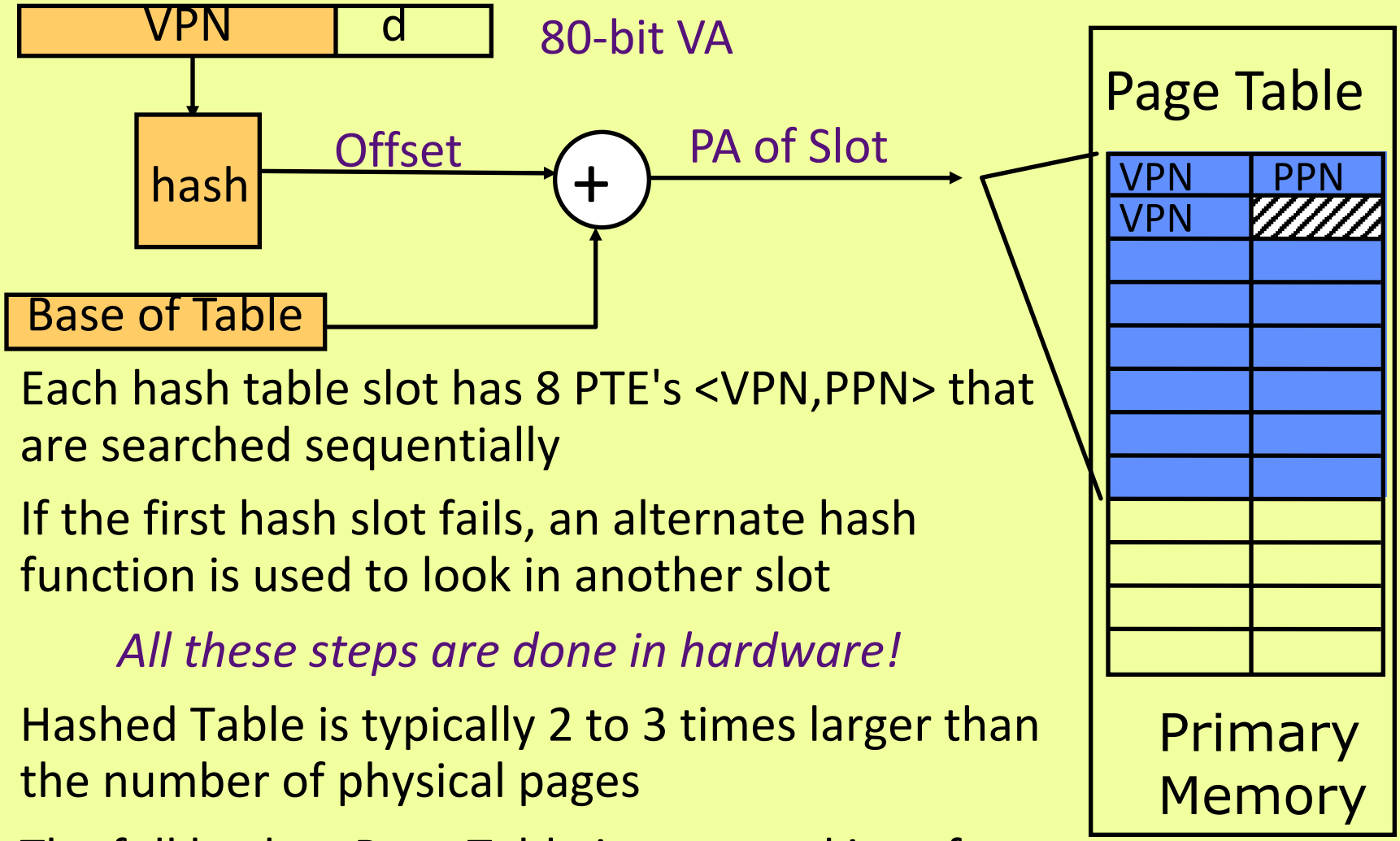


Hashed Page Table: Approximating Associative Addressing



- Hashed Page Table is typically 2 to 3 times larger than the number of PPN's to reduce collision probability
- It can also contain DPN's for some non-resident pages (not common)
- If a translation cannot be resolved in this table then the software consults a data structure that has an entry for every existing page (e.g., full page table)

Power PC: Hashed Page Table



- Each hash table slot has 8 PTE's <VPN,PPN> that are searched sequentially
- If the first hash slot fails, an alternate hash function is used to look in another slot

All these steps are done in hardware!

- Hashed Table is typically 2 to 3 times larger than the number of physical pages
- The full backup Page Table is managed in software

RISC-V Privilege Modes

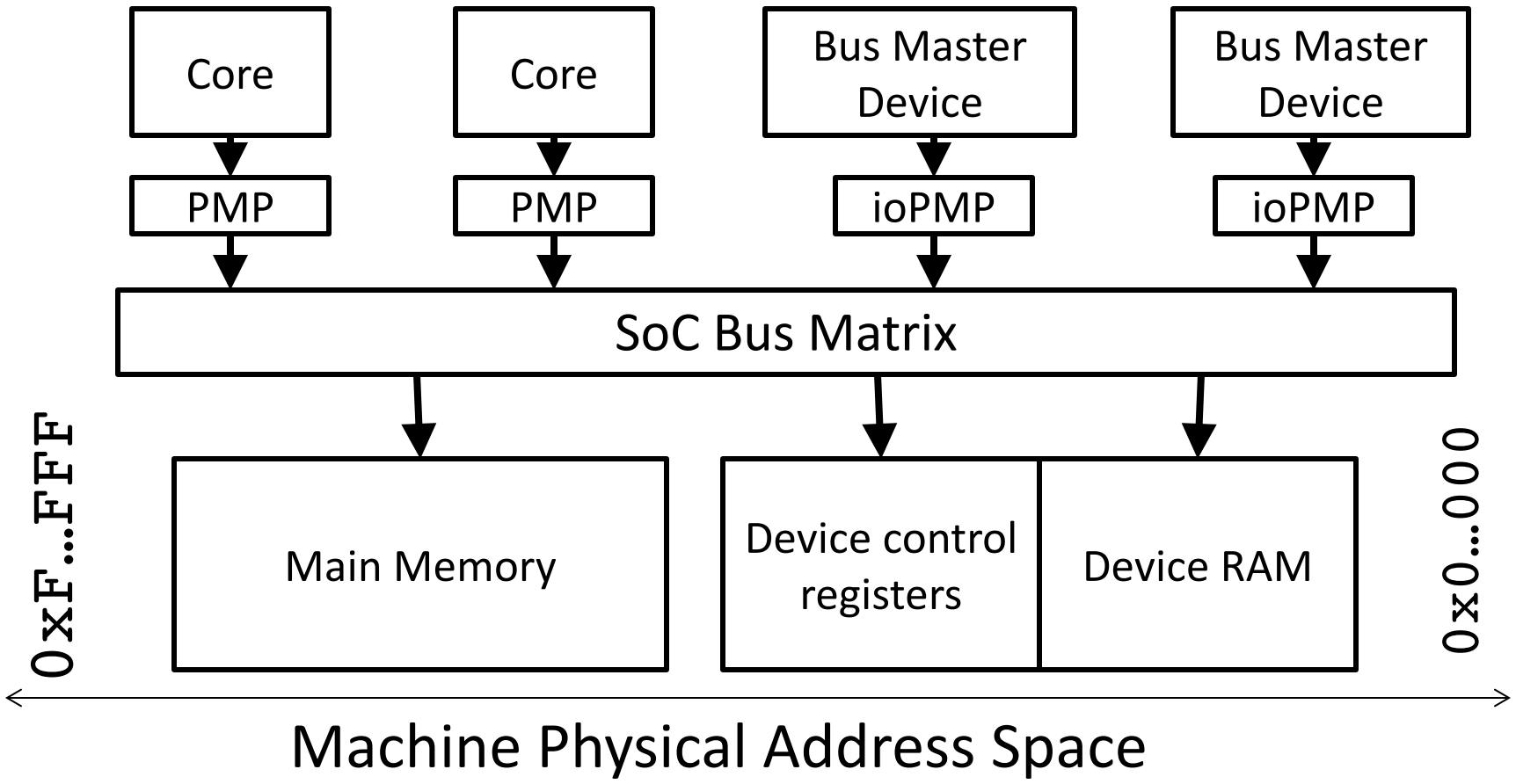
- Machine mode (M-mode)
  - AKA monitor mode, microcode mode, …
- Hypervisor-Extended Supervisor Mode (HS-Mode)
- Supervisor Mode (S-mode)
- User Mode (U-mode)

- Supported combinations of modes:
  - M (simple embedded systems)
  - M, U (embedded systems with security)
  - M, S, U (systems running Unix-like operating systems)
  - M, S, HS, U (systems running hypervisors)

RISC-V System State

- Processor registers
  - Compute registers
    - General-purpose (x0-x31)
    - Optional floating-point (f0-f31)
    - Optional vector (v0-v31)
    - Optional custom
  - Control and status registers (CSRs)
    - Accessibility controlled by privilege mode
- System main memory
- System I/O devices
- All system memory and device control registers mapped into flat machine physical address space

Physical Memory Protection (PMP)



M-Mode controls PMPs

- M-mode has access to entire machine after reset
- Configures PMPs and ioPMPs to contain each active context inside a physical partition
- Can even restrict M-mode access to regions until next reset
- M-mode can dynamically swap PMP settings to run different security contexts on a hart

RISC-V PMP Configuration

7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Figure 3.27: PMP configuration register format.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

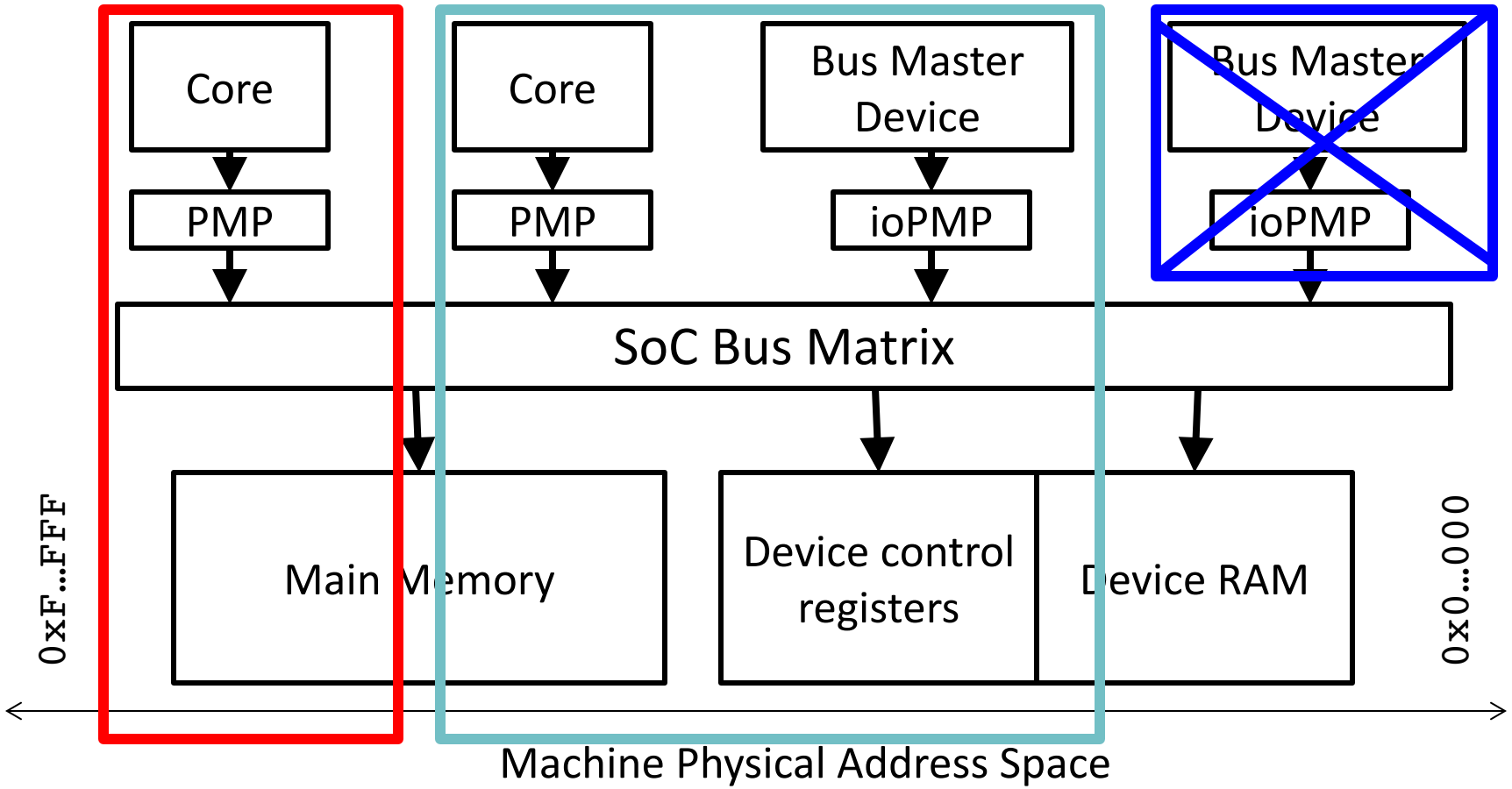
Table 3.8: Encoding of A field in PMP configuration registers.

pmpaddr	pmpcfg.A	Match type and size
yyyy...yyyy	NA4	4-byte NAPOT range
yyyy...yyy0	NAPOT	8-byte NAPOT range
yyyy...yy01	NAPOT	16-byte NAPOT range
yyyy...y011	NAPOT	32-byte NAPOT range
...	...	...
yy01...1111	NAPOT	2^{XLEN}-byte NAPOT range
y011...1111	NAPOT	2^{XLEN+1}-byte NAPOT range
0111...1111	NAPOT	2^{XLEN+2}-byte NAPOT range
1111...1111	NAPOT	*Reserved*

Table 3.9: NAPOT range encoding in PMP address and configuration registers.

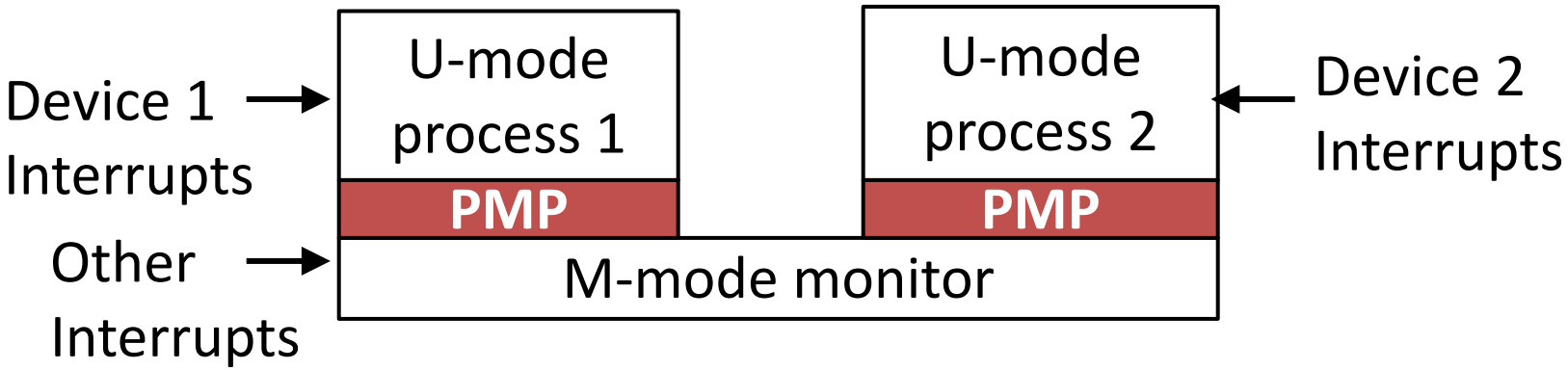
NAPOT = Naturally Aligned Power-of-2

Multiple Concurrent Security Contexts



RISC-V Secure Embedded Systems (M, U modes)

- M-mode runs secure boot and runtime monitor
- Embedded code runs in U-mode
- Physical memory protection (PMP) on U-mode accesses
- Interrupt handling can be delegated to U-mode code
  - User-level interrupt support (N-extension)
- Provides arbitrary number of isolated security contexts

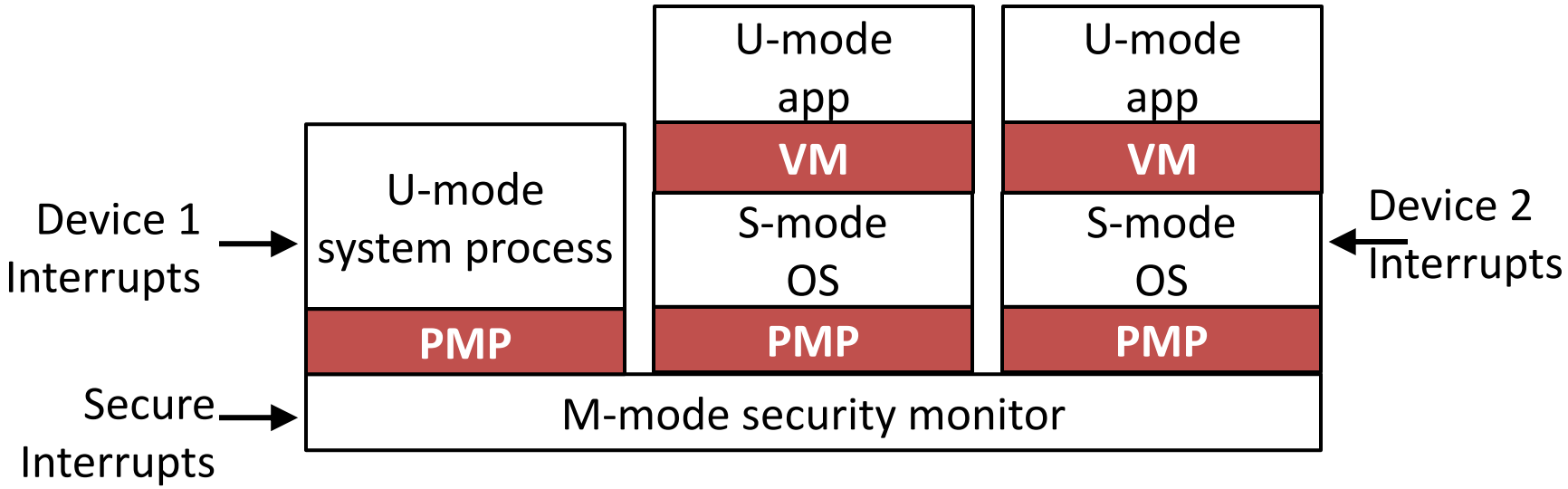


RISC-V Virtual Memory Architectures (M, S, U modes)

- Designed to support current Unix-style operating systems
- Sv32 (RV32)
  - Demand-paged 32-bit virtual-address spaces
  - 2-level page table
  - 4 KiB pages, 4 MiB megapages
- Sv39 (RV64)
  - Demand-paged 39-bit virtual-address spaces
  - 3-level page table
  - 4 KiB pages, 2 MiB megapages, 1 GiB gigapages
- Sv48, Sv57, Sv64 (RV64)
  - Sv39 + 1/2/3 more page-table levels

S-Mode runs on top of M-mode

- M-mode runs secure boot and monitor
- S-mode runs OS
- U-mode runs application on top of OS or M-mode



VM features track historical uses:

- Bare machine, only physical addresses
  - One program owned entire machine
- Batch-style multiprogramming
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (supports *swapping* entire programs but not demand-paged virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived
- Time sharing
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory
- Virtual Machine Monitors
  - Run multiple operating systems on one machine
  - Idea from 1970s IBM mainframes, now common on laptops
    - e.g., run Windows on top of Mac OS X
  - Hardware support for two levels of translation/protection
    - Guest OS virtual -> Guest OS physical -> Host machine physical

Virtual Memory Use Today - 1

- Servers/desktops/laptops/smartphones have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Vector supercomputers have translation and protection but rarely complete demand-paging
- (Older Crays: base&bound, Japanese & Cray X1/X2: pages)
  - Don't waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions

Virtual Memory Use Today - 2

- Most embedded processors and DSPs provide physical addressing only
  - Can't afford area/speed/power budget for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)