EE 660: Computer Architecture
Address Translation and Protection

Yao Zheng

Department of Electrical Engineering

University of Hawai‘i at Mānoa

UNIVERSITY of HAWAI‘I®
MĀNOA

Based on the slides of Prof. David Wentzlaff

Memory Management

- From early absolute addressing schemes, to modern virtual memory systems with support for virtual machine monitors

- Can separate into orthogonal functions:
  - Translation (mapping of virtual address to physical address)
  - Protection (permission to access word in memory)
  - Virtual Memory (transparent extension of memory space using slower disk storage)
- But most modern systems provide support for all the above functions with a single page-based system

Absolute Addresses

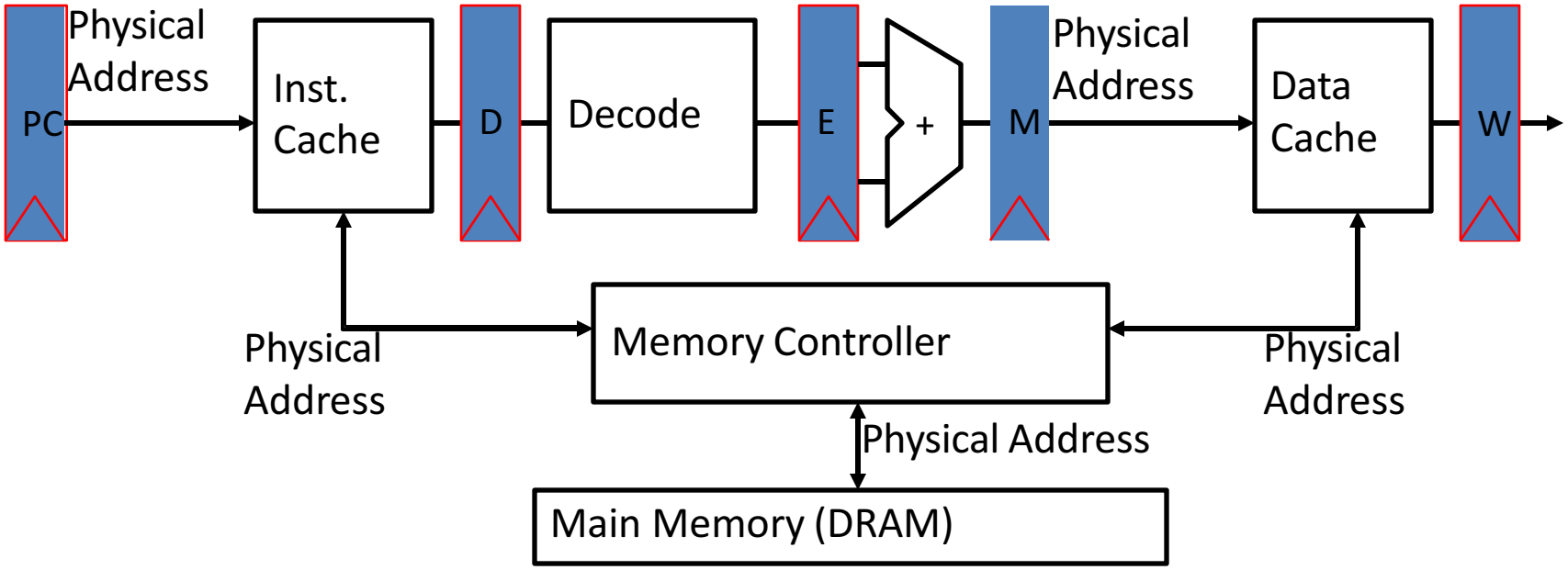
EDSAC, early 50's

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

How could location independence be achieved?

Linker and/or loader modify addresses of subroutines and callers when building a program memory image

Bare Machine



- In a bare machine, the only kind of address is a physical address

Dynamic Address Translation

Location-independent programs

Programming and storage management ease

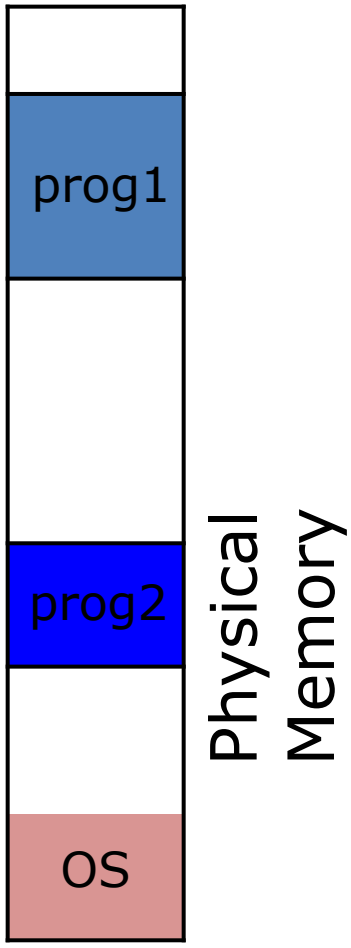
⇒ need for a *base register*

Protection

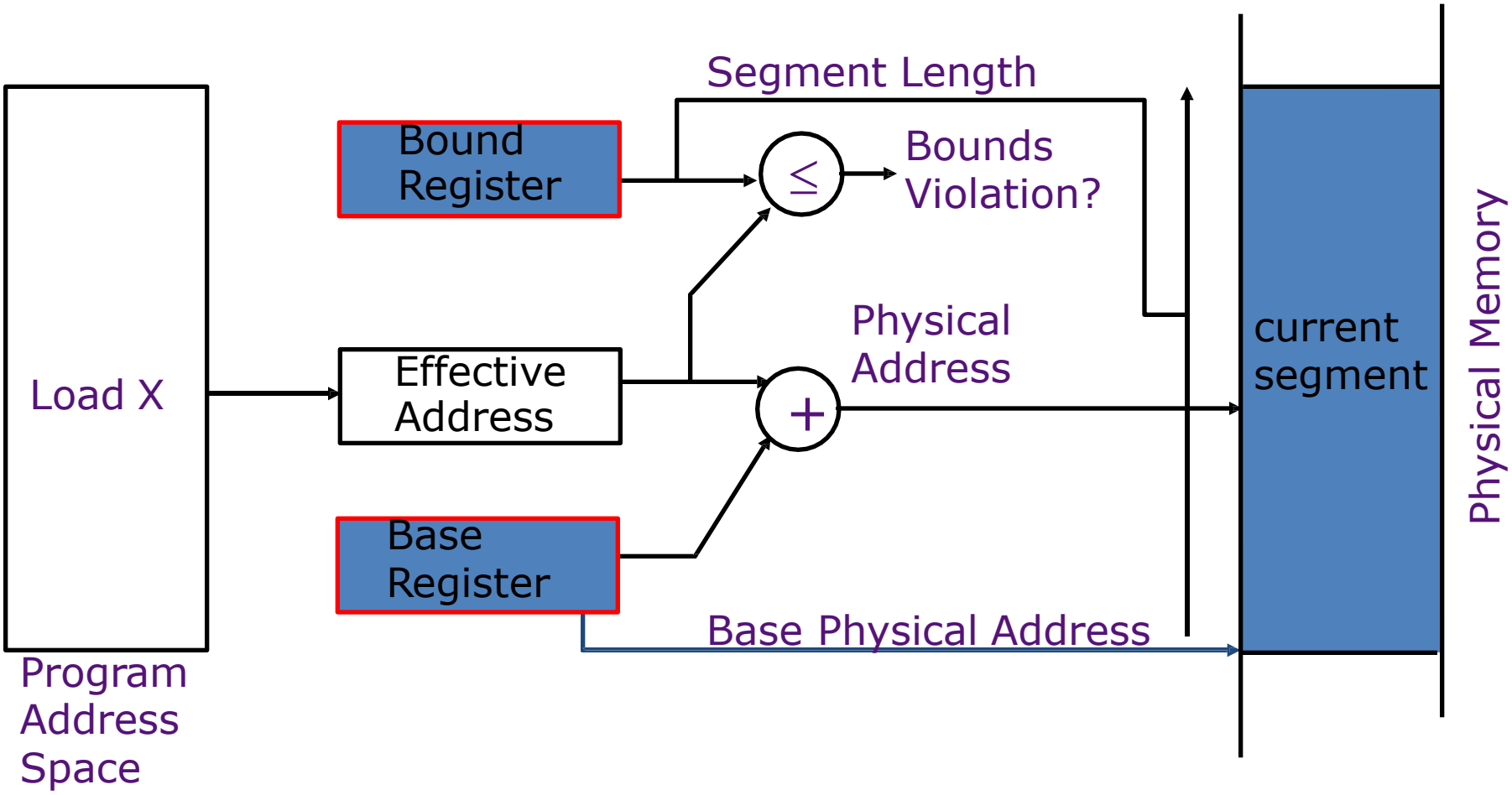
Independent programs should not affect each other inadvertently

⇒ need for a *bound register*

Multiprogramming drives requirement for resident *supervisor* to manage context switches between multiple programs

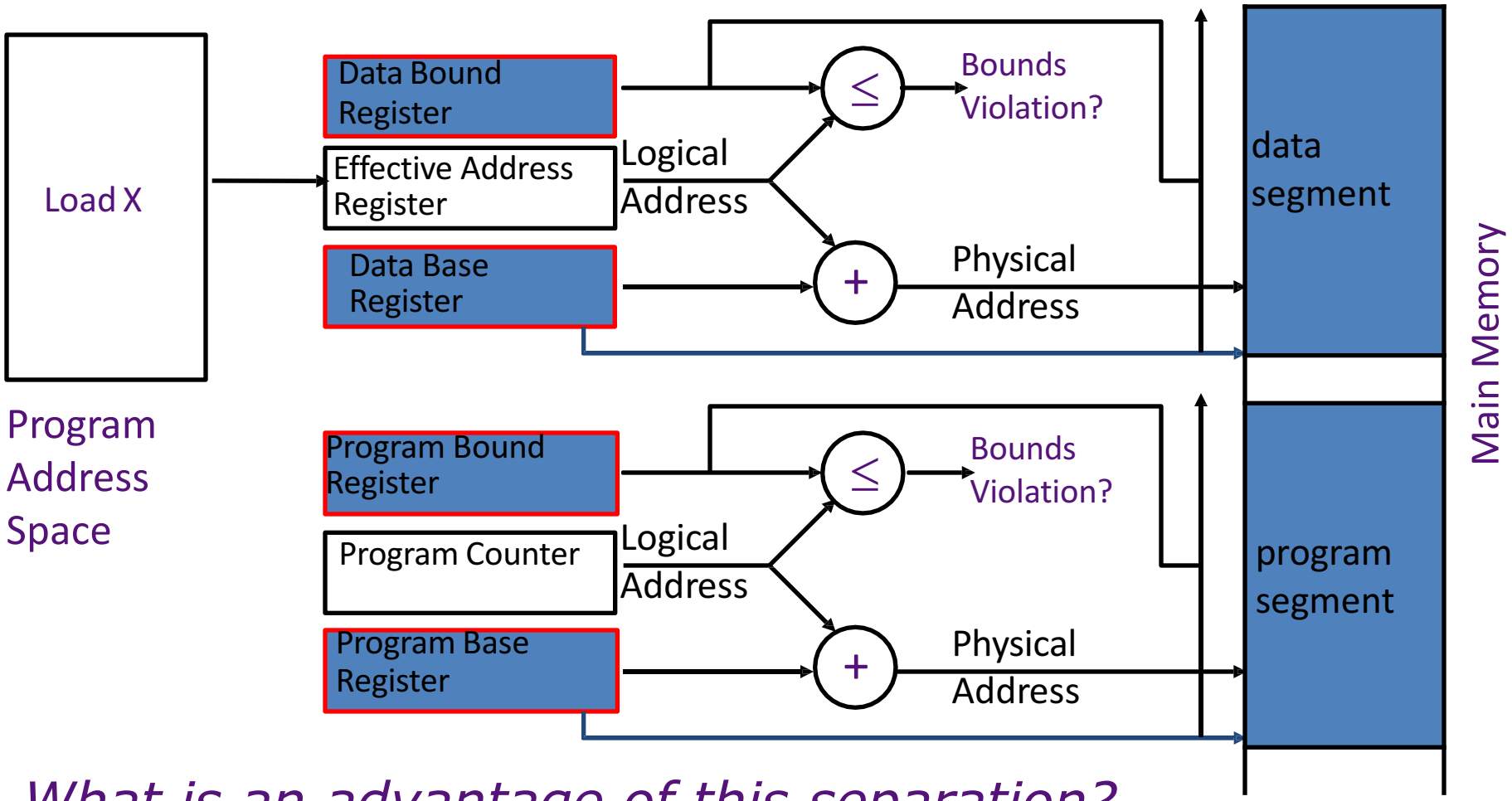


Simple Base and Bound Translation



Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*

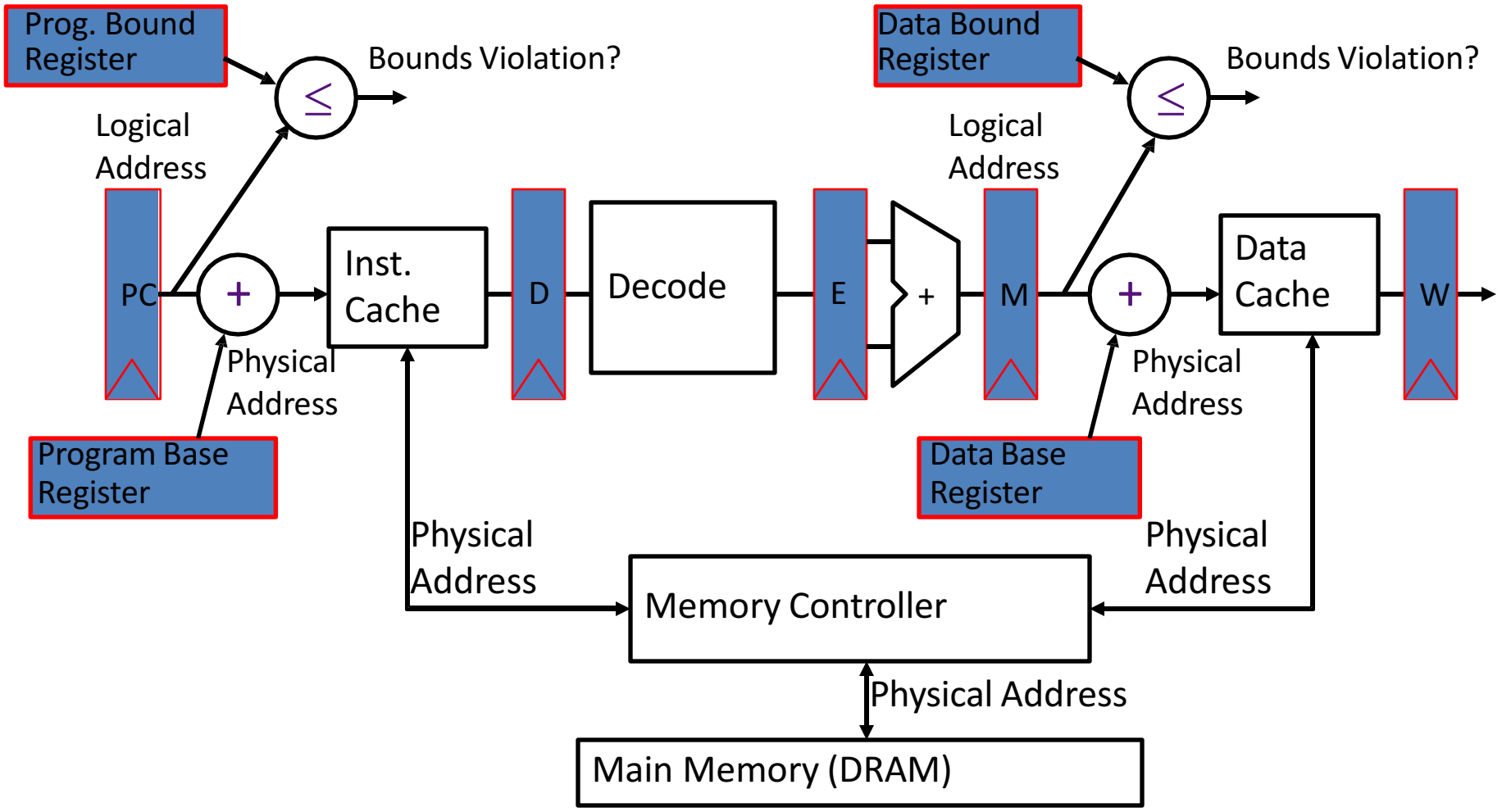
Separate Areas for Program and Data



What is an advantage of this separation?

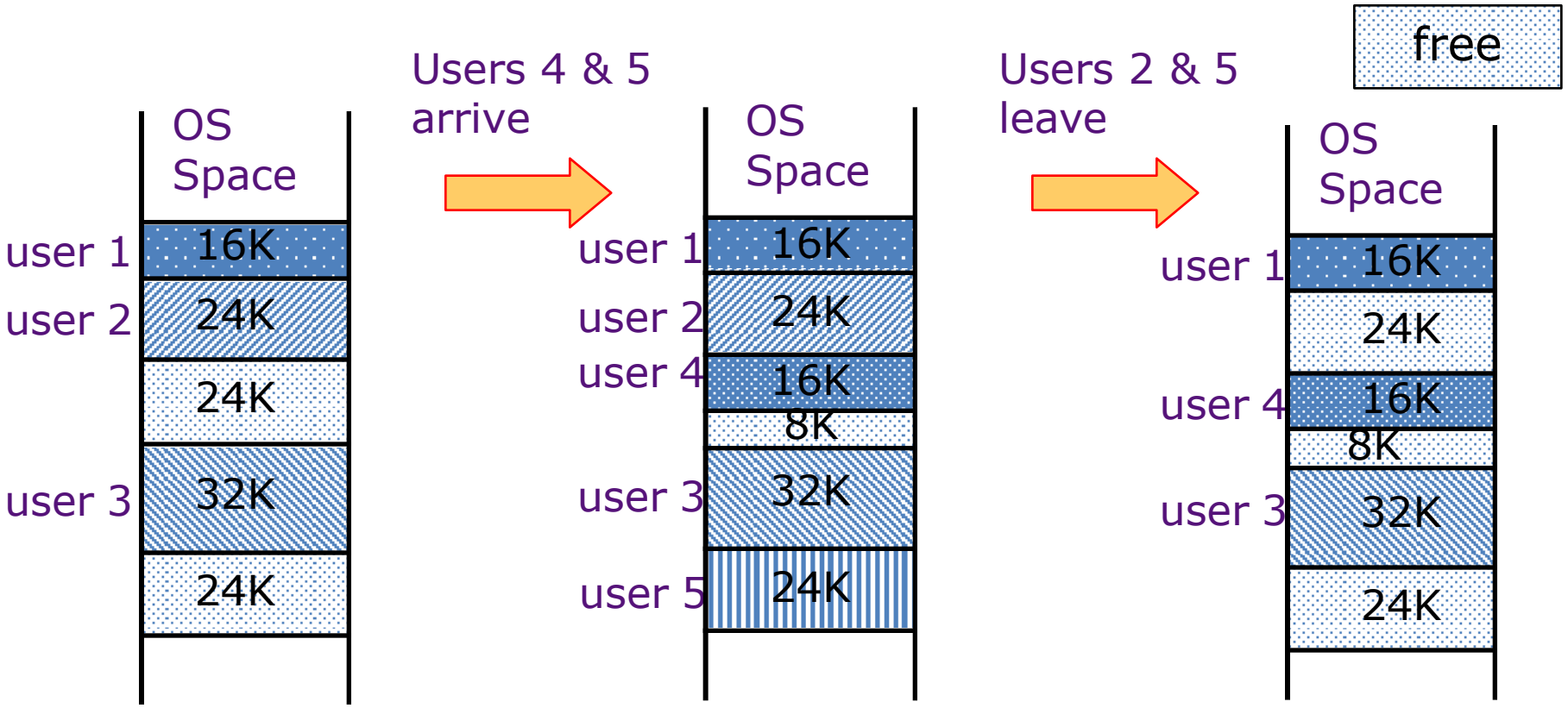
(Scheme used on all Cray vector supercomputers prior to X1, 2002)

Base and Bound Machine



[Can fold addition of base register into (base+offset) calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)]

Memory Fragmentation



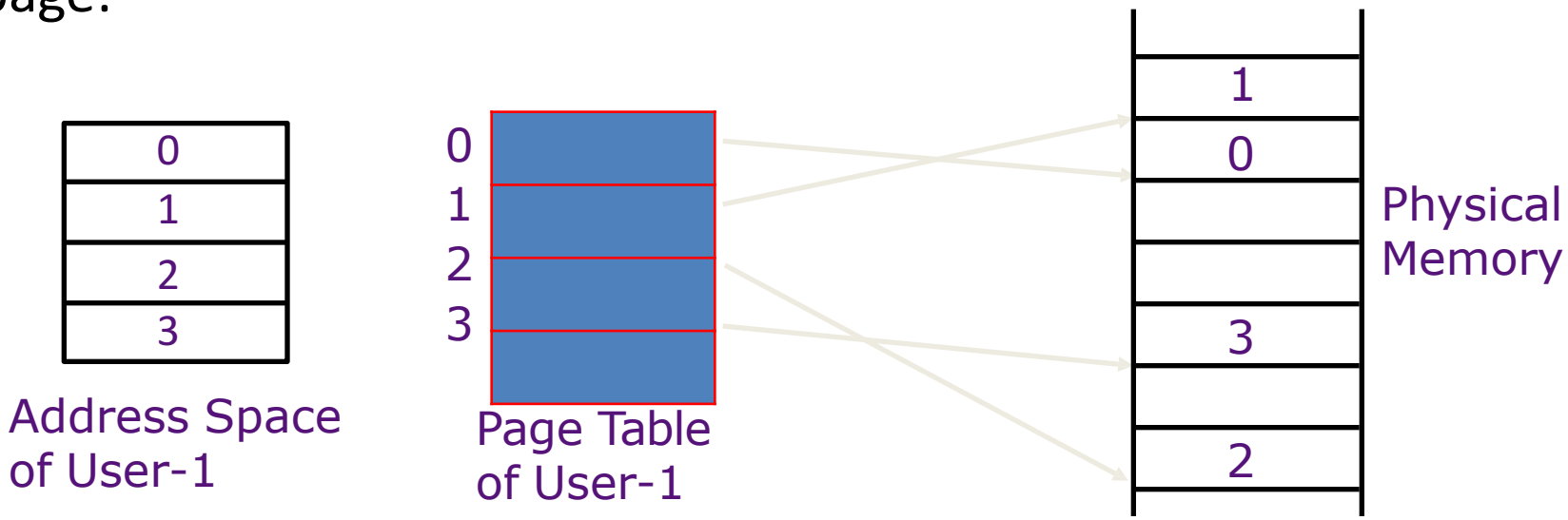
As users come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage.

Paged Memory Systems

- Processor-generated address can be interpreted as a pair <page number, offset>:

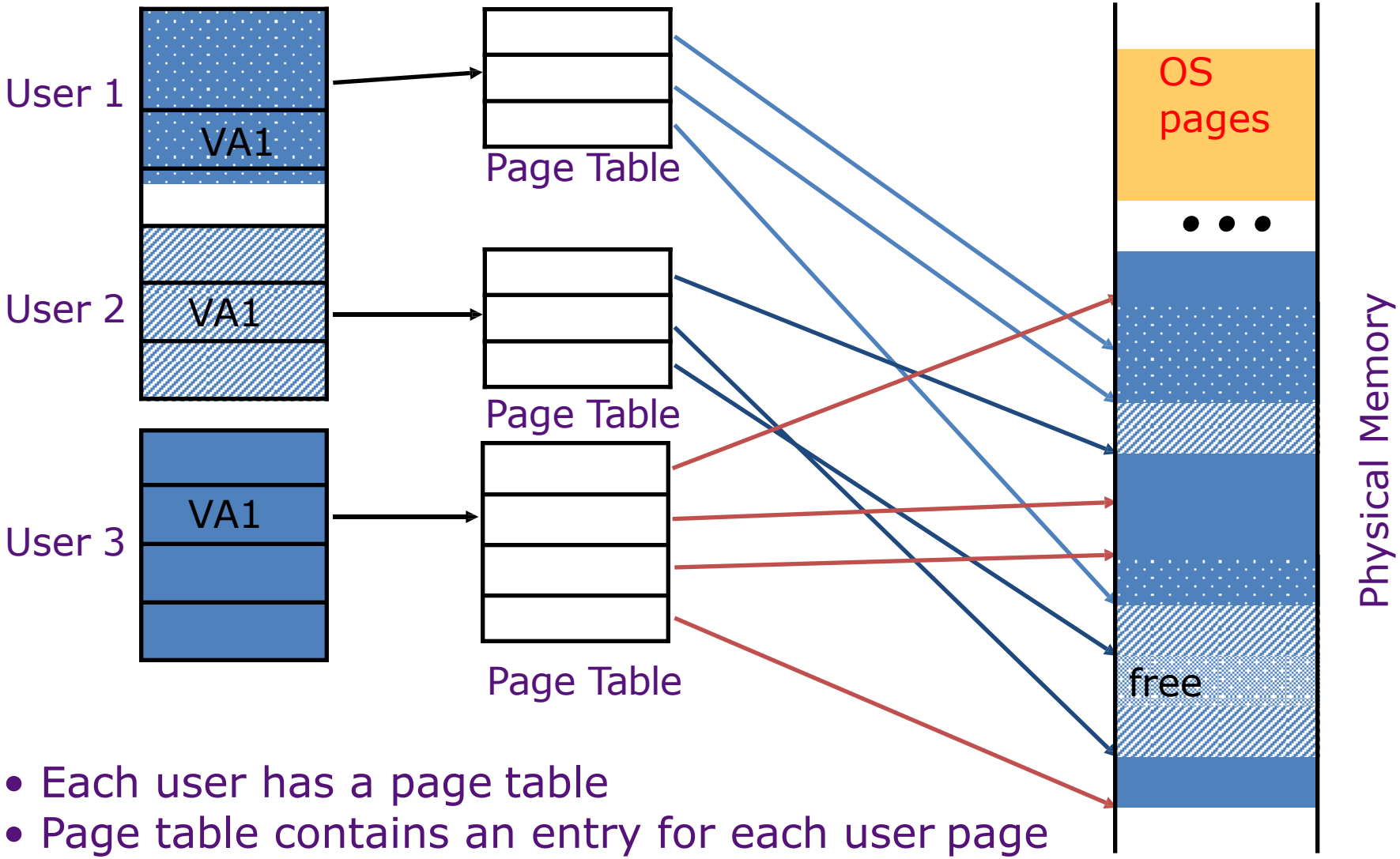
page number	offset

- A page table contains the physical address of the base of each page:



Page tables make it possible to store the pages of a program non-contiguously.

Private Address Space per User

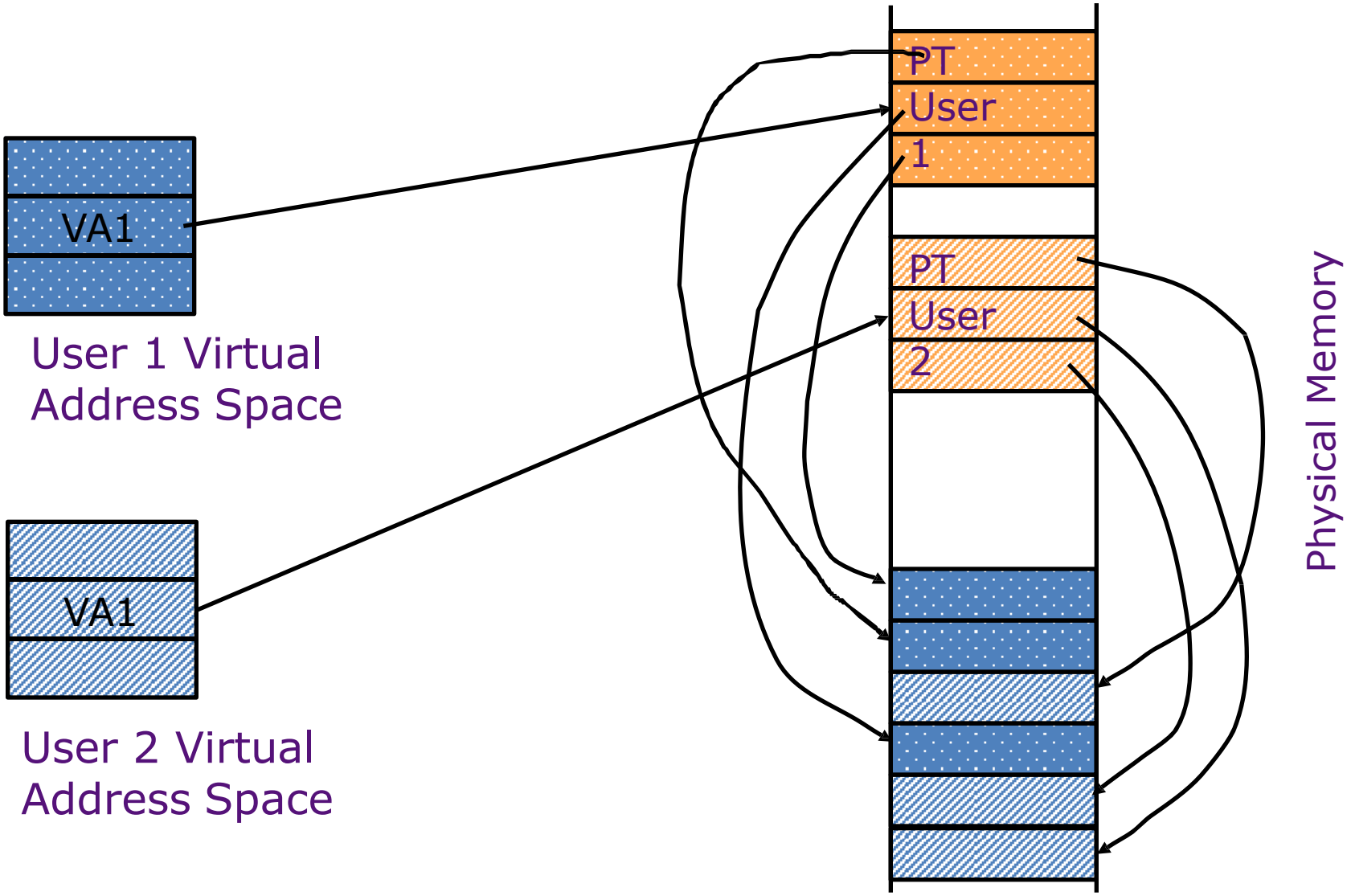


- Each user has a page table
- Page table contains an entry for each user page

Where Should Page Tables Reside?

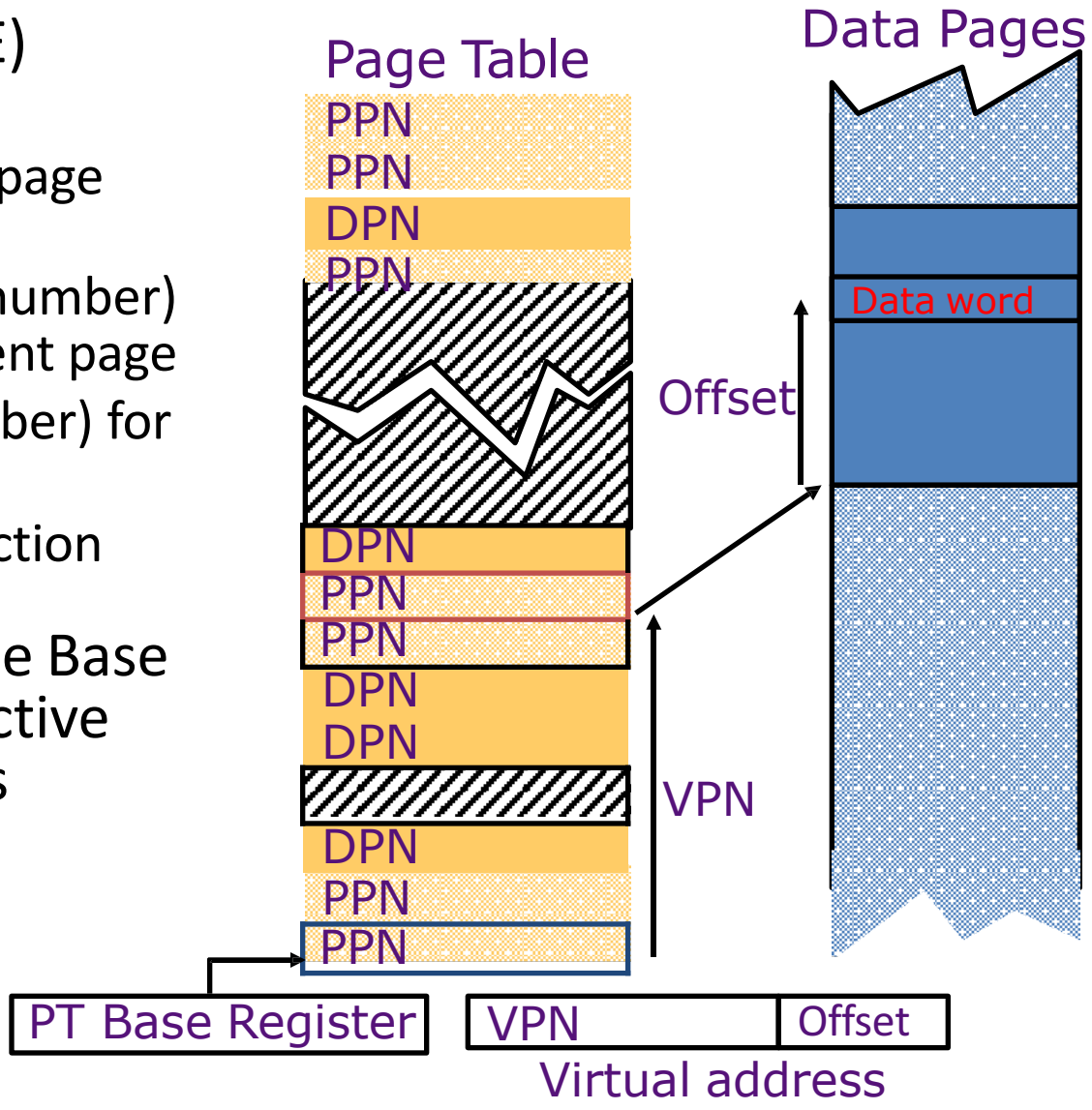
- Space required by the page tables (PT) is proportional to the address space, number of users, (inverse to) size of each page, ...
  - Space requirement is large
  - Too expensive to keep in registers
- Idea: Keep PTs in the main memory
  - needs one reference to retrieve the page base address and another to access the data word
    - *doubles the number of memory references!*
  - Storage space to store PT grows with size of memory

Page Tables in Physical Memory



Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage
- OS sets the Page Table Base Register whenever active user process changes



Size of Linear Page Table

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒ 2^{20} PTEs, i.e, 4 MB page table per user per process

⇒ 4 GB of swap needed to back up full virtual address space

Larger pages?

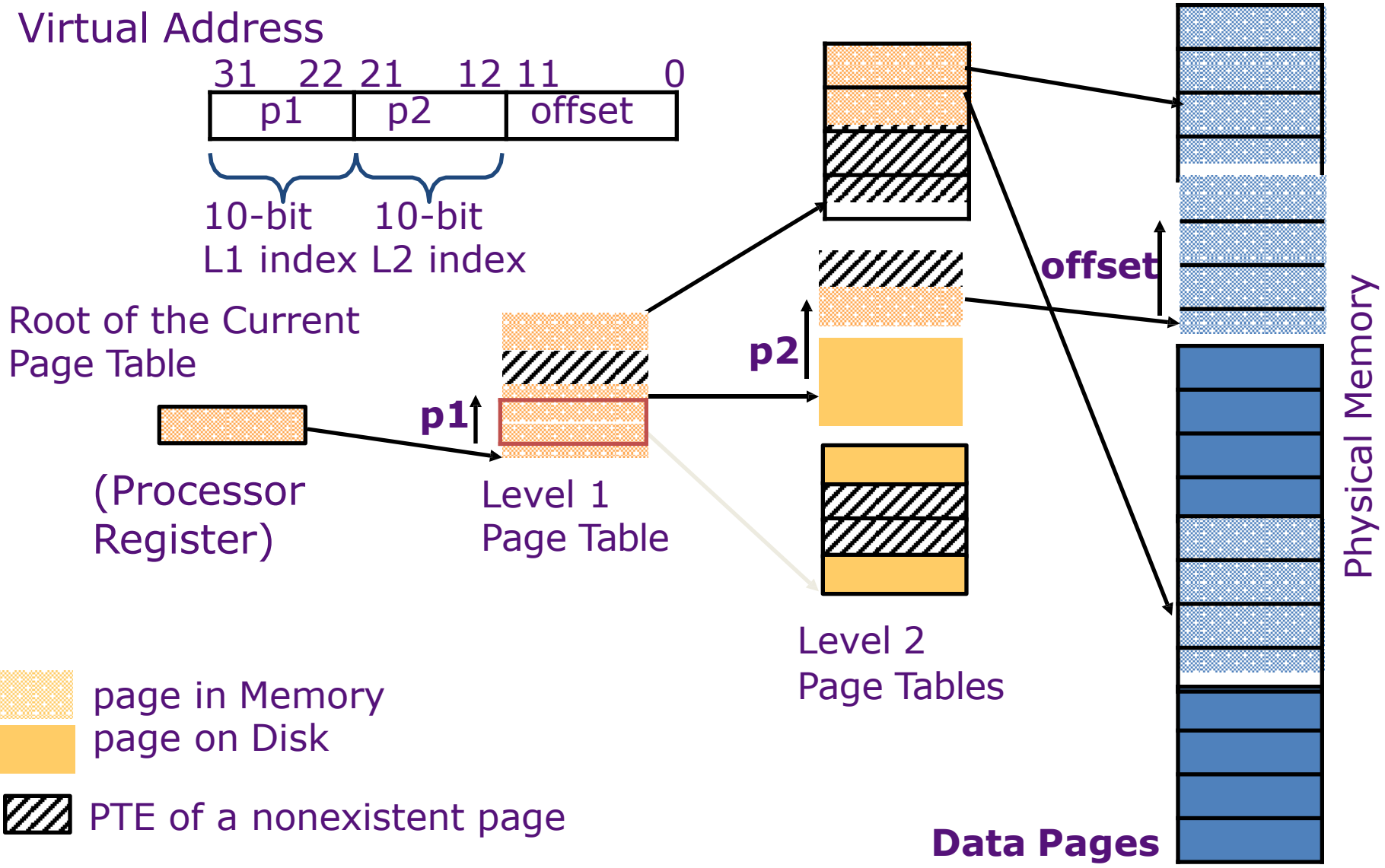
- Internal fragmentation (Not all memory in page is used)
- Larger page fault penalty (more time to read from disk)

What about 64-bit virtual address space???

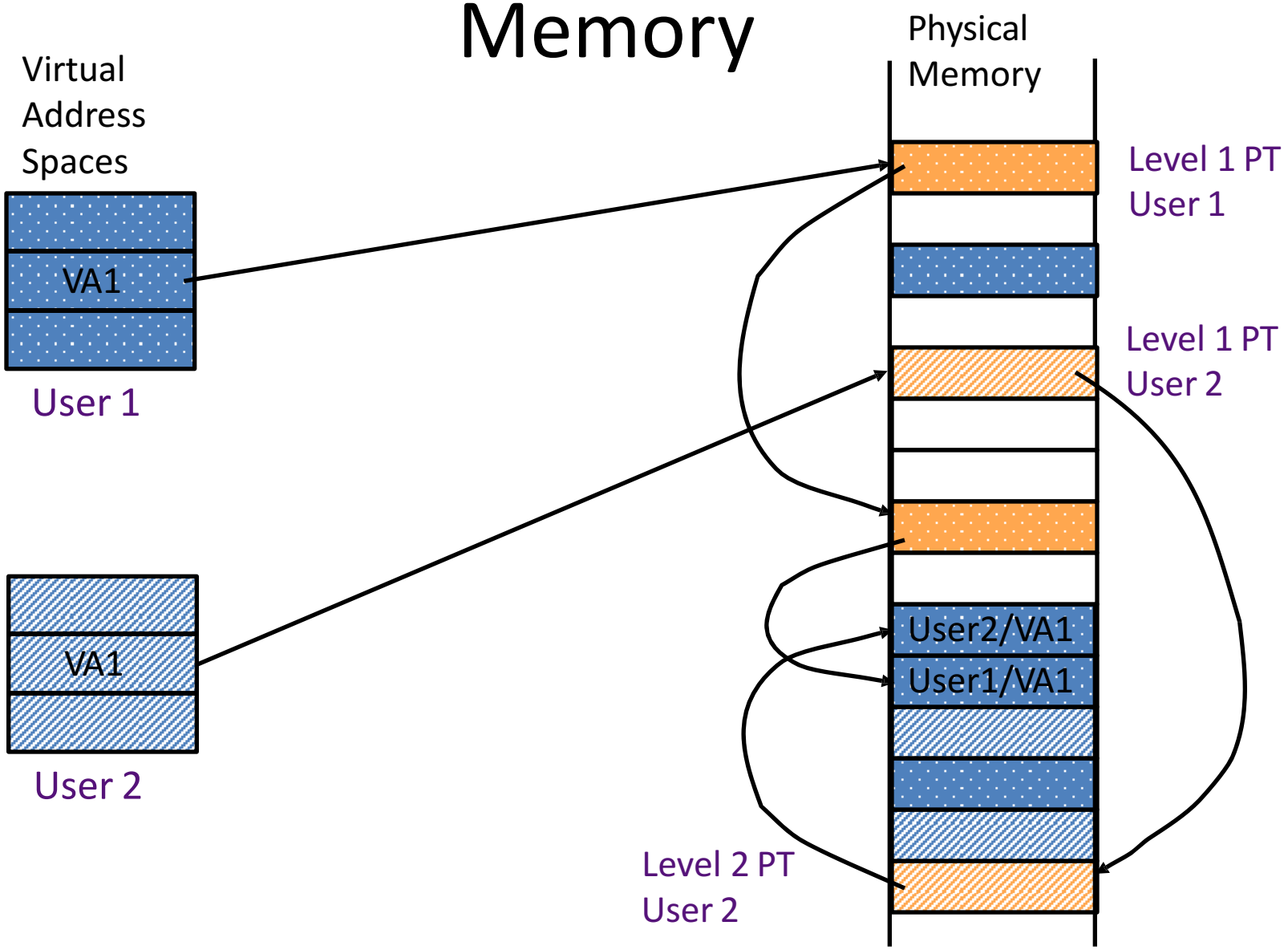
- Even 1MB pages would require 2^{44} 8-byte PTEs (35 TB!)

What is the "saving grace" ?

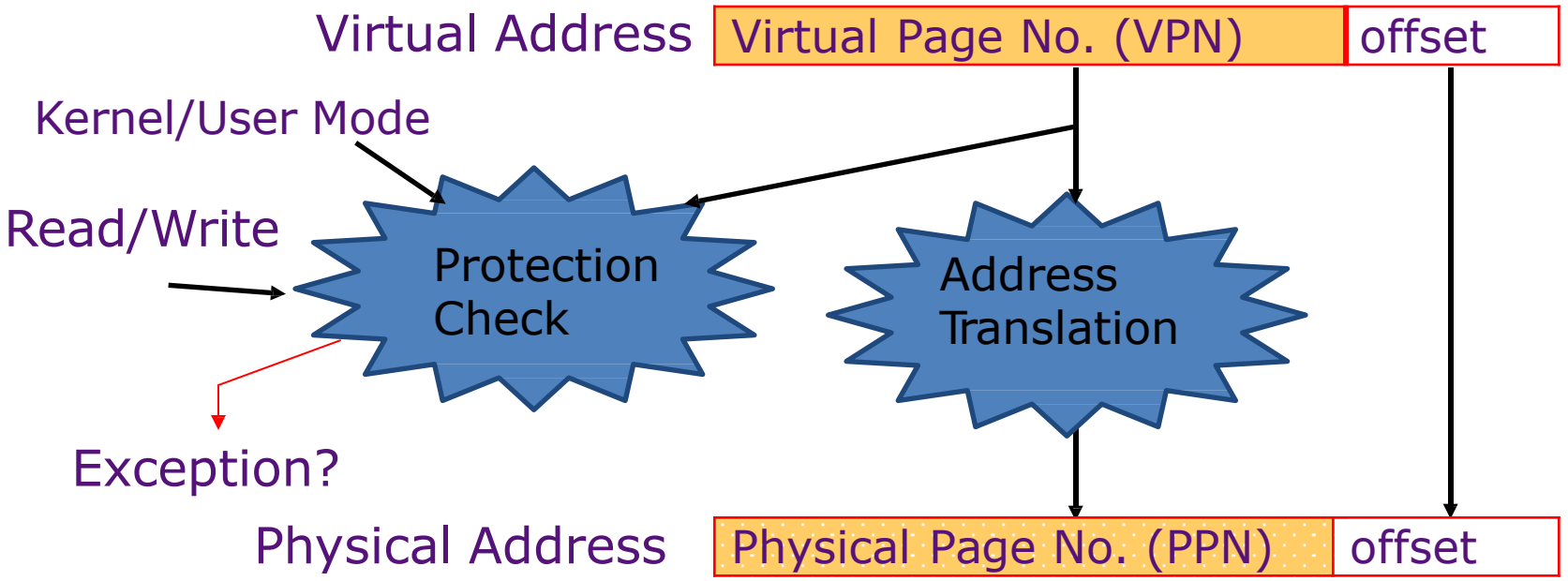
Hierarchical Page Table



Two-Level Page Tables in Physical Memory



Address Translation & Protection



- Every instruction and data access needs address translation and protection checks

A good Virtual Memory (VM) design needs to be fast (~ one cycle) and space efficient

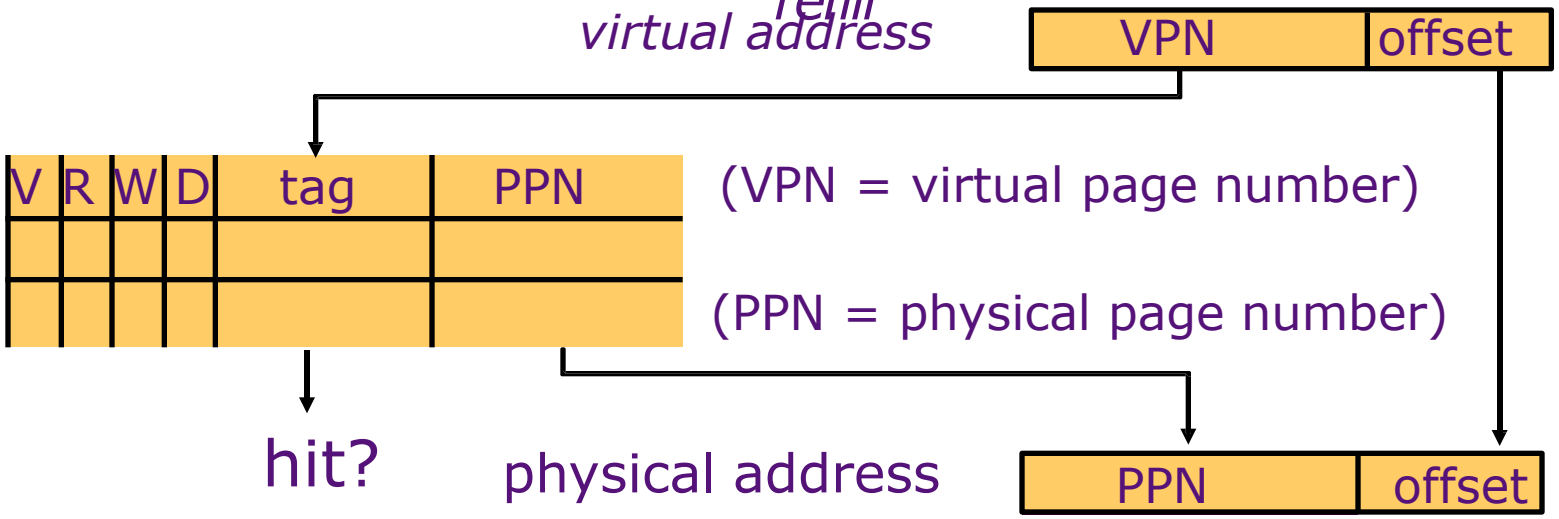
Translation Lookaside Buffers (TLB)

Problem: Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single-Cycle Translation*

TLB miss ⇒ *Page-Table Walk to refill*



TLB Designs

- Typically 16-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages ➔ more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random (Clock Algorithm) or FIFO replacement policy
- No process information in TLB
  - Flush TLB on Process Context Switch
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

  Example: 64 TLB entries, 4KB pages, one page per entry

  TLB Reach =_____*?*

TLB Designs

- Typically 16-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages ➔ more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random (Clock Algorithm) or FIFO replacement policy
- No process information in TLB
  - Flush TLB on Process Context Switch
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

  Example: 64 TLB entries, 4KB pages, one page per entry

  TLB Reach = 64 entries * 4KB = 256KB (if contiguous)?

TLB Extensions

- Address Space Identifier (ASID)
  - Allow TLB Entries from multiple processes to be in TLB at same time. ID of address space (Process) is matched on.
  - Global Bit (G) can match on all ASIDs
- Variable Page Size (PS)
  - Can increase reach on a per page basis

V	R	W	D	tag	PPN	PS	G	ASID

Handling a TLB Miss

Software (MIPS, Alpha)

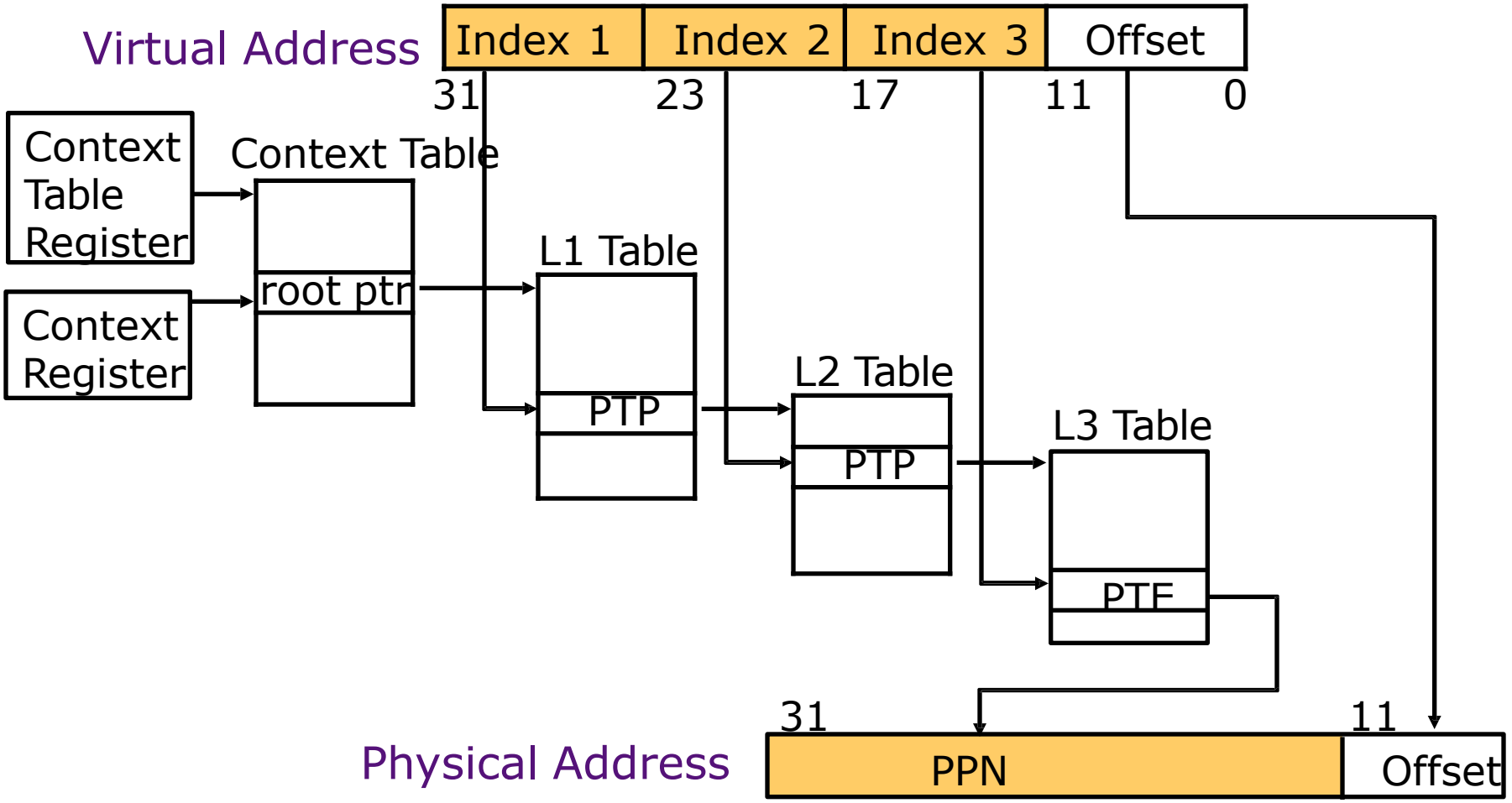
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

Hardware (SPARC v8, x86, PowerPC)

A memory management unit (MMU) walks the page tables and reloads the TLB

If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

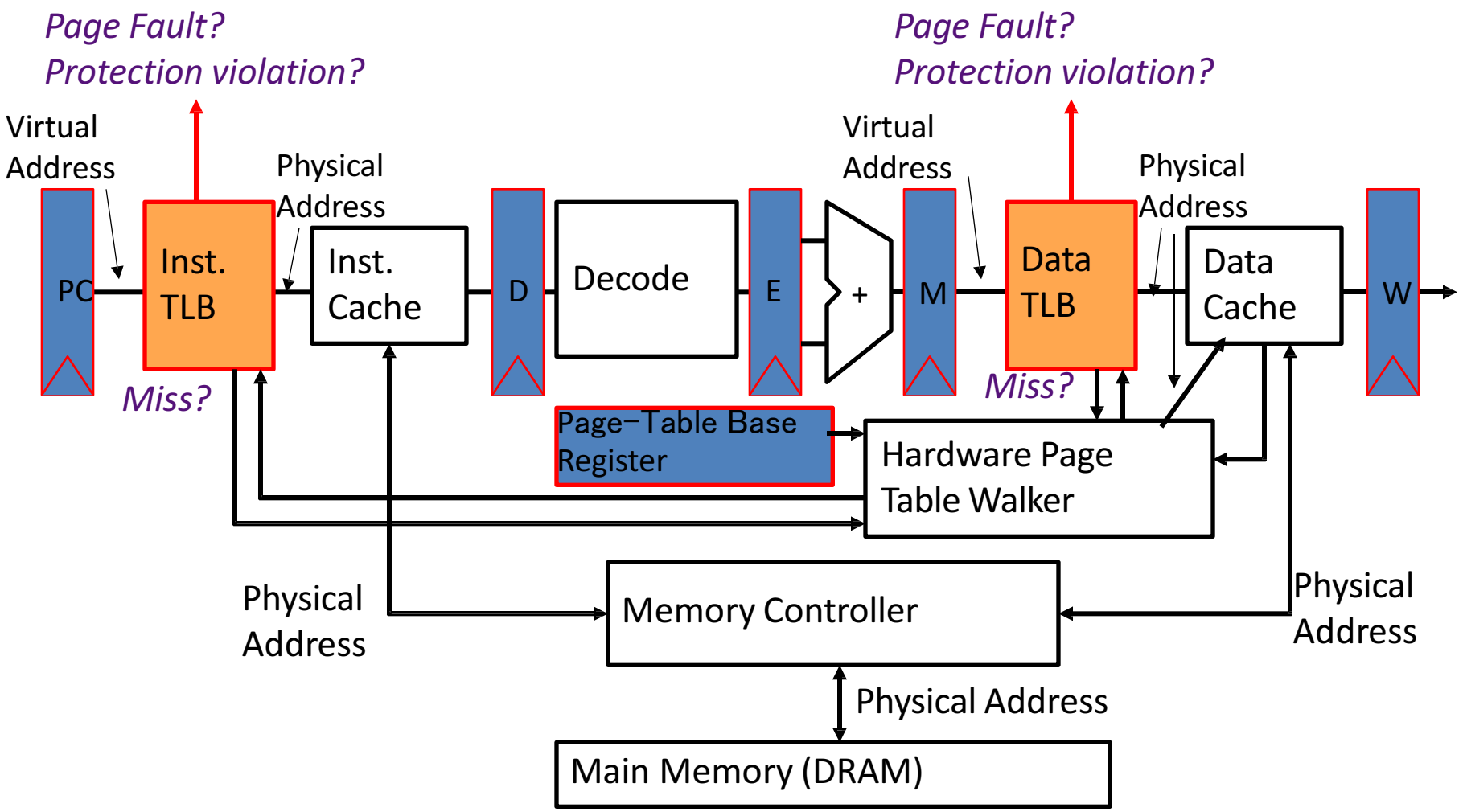
Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

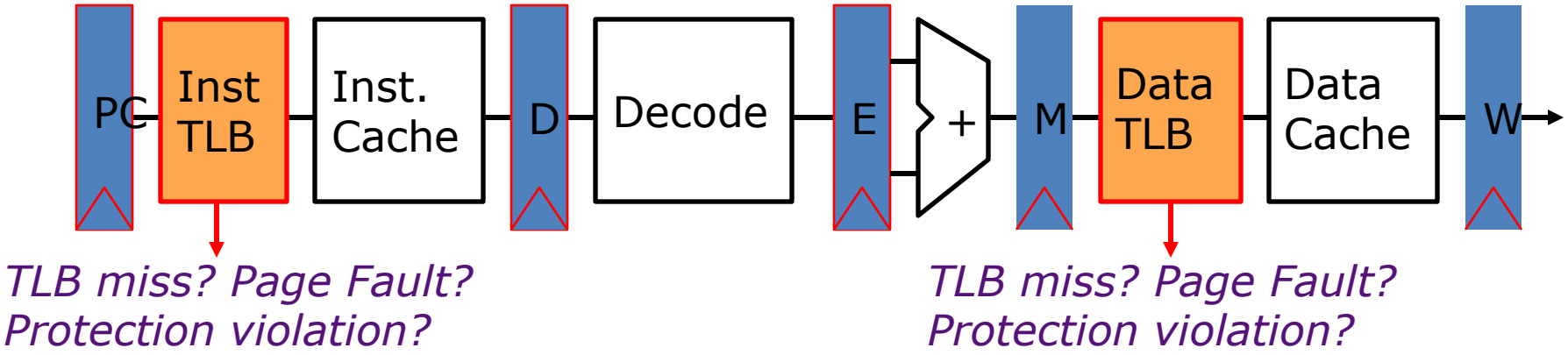
Page-Based Virtual-Memory Machine

(Hardware Page-Table Walk)



- **Assumes page tables held in untranslated physical memory**

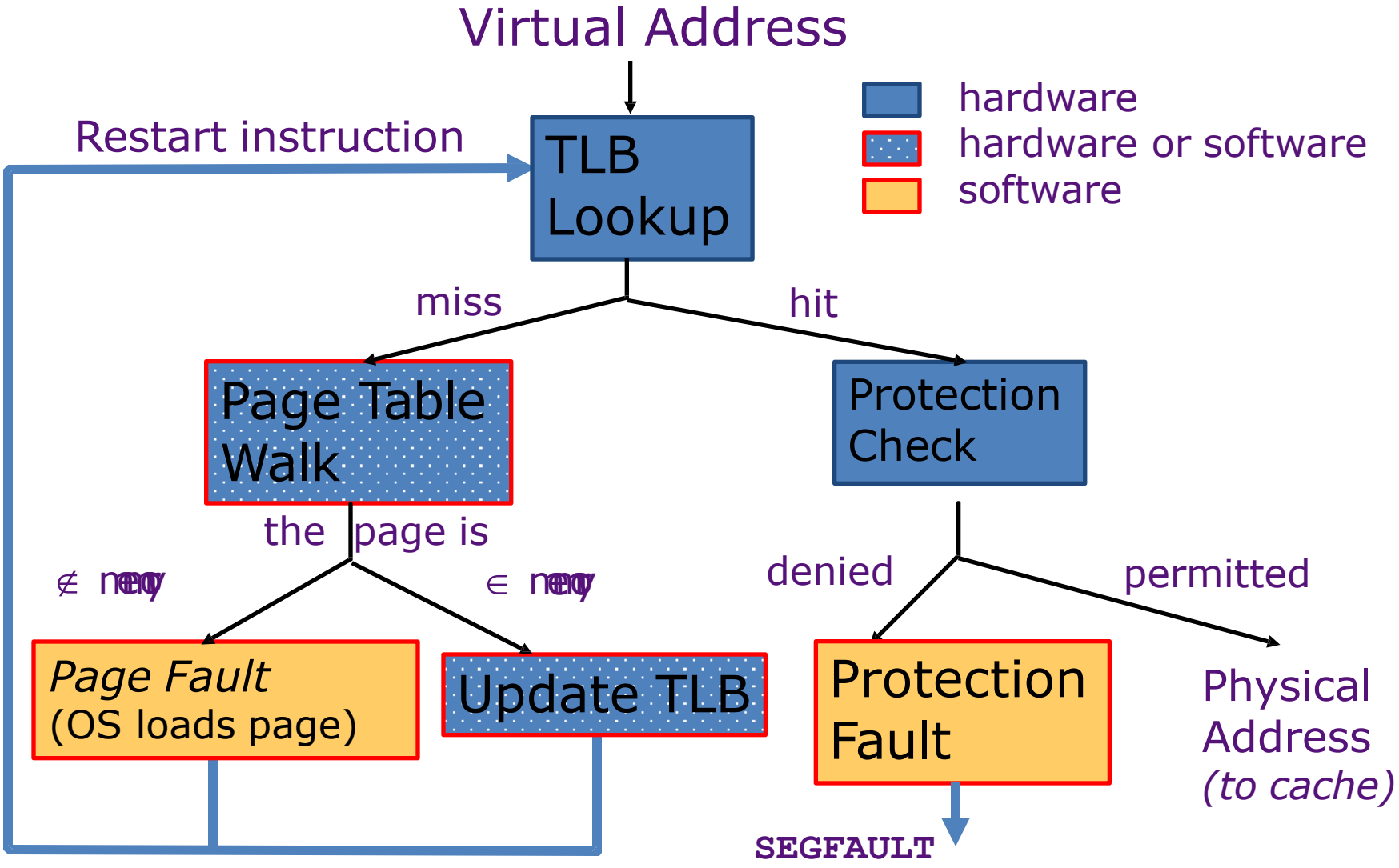
Address Translation in CPU Pipeline



- Software handlers need *restartable* exception on TLB fault
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- 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

Address Translation:

putting it all together



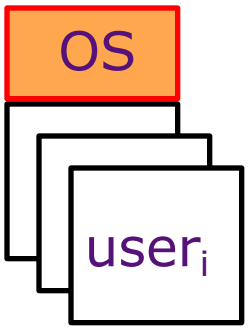
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

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

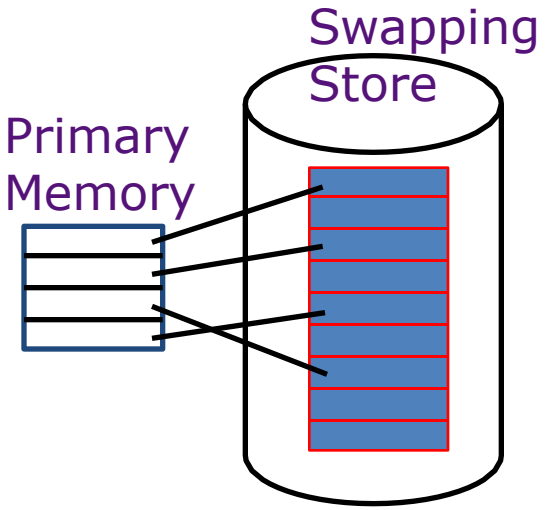
page table ≡name space



Demand Paging

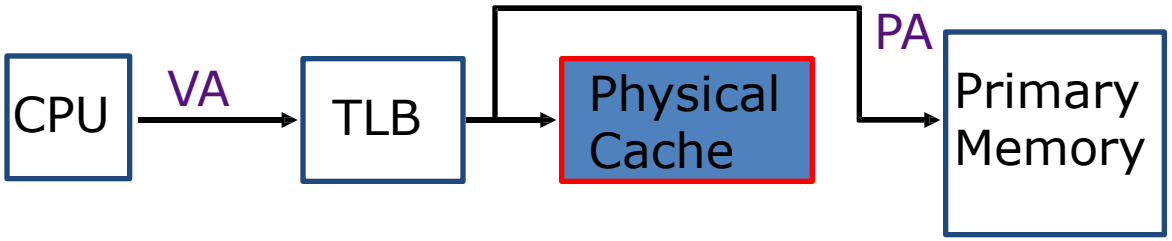
Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

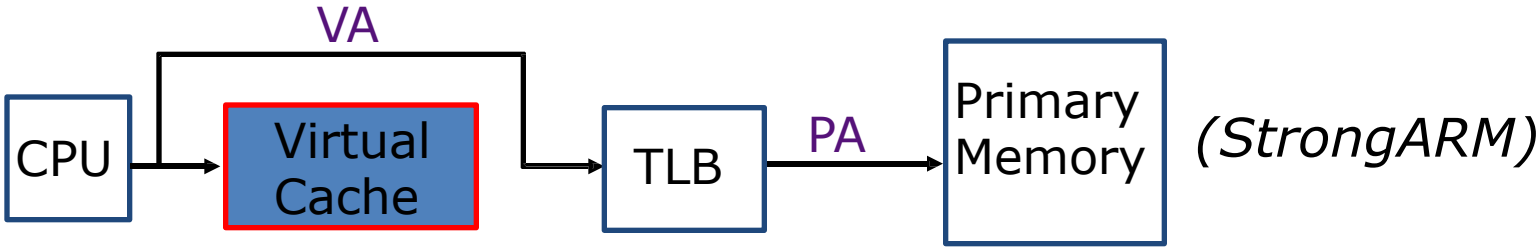


The price is address translation on each memory reference

Virtual-Address Caches



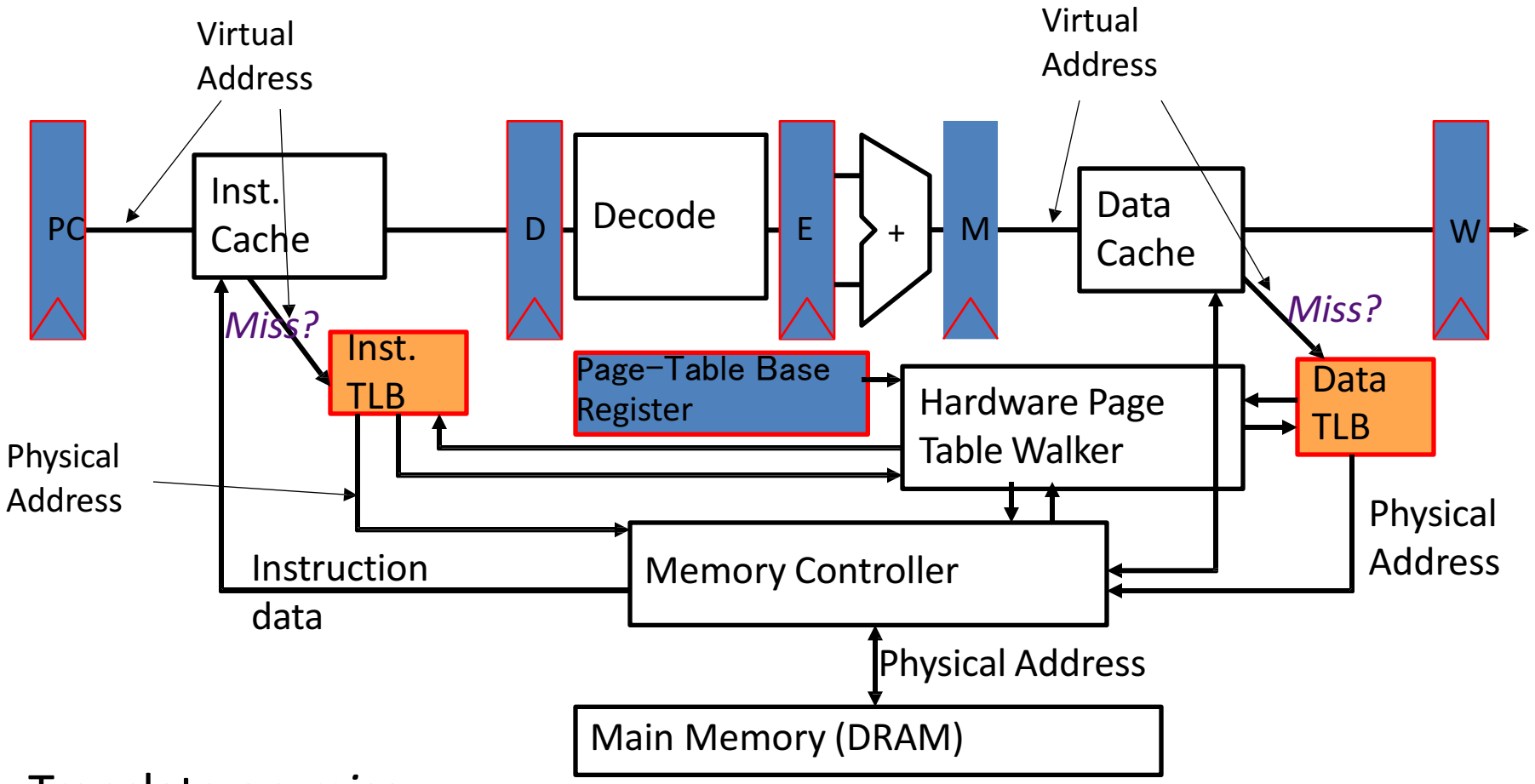
Alternative: place the cache before the TLB



(StrongARM)

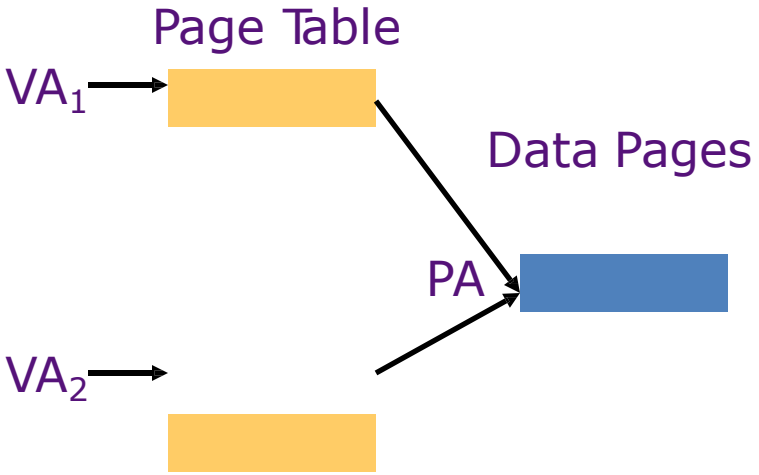
- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- *aliasing problems* due to the sharing of pages (-)
- maintaining cache coherence (-) (*see later in course*)

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 P^A

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)

Cache-TLB Interactions

- Physically Indexed/Physically Tagged
- Virtually Indexed/Virtually Tagged
- Virtually Indexed/Physically Tagged
  - Concurrent cache access with TLB Translation
- Both Indexed/Physically Tagged
  - Small enough cache or highly associative cache will have fewer indexes than page size
  - Concurrent cache access with TLB Translation
- ~~Physically Indexed/Virtually Tagged~~

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)

- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750