

# Address Translation and Virtual Memory

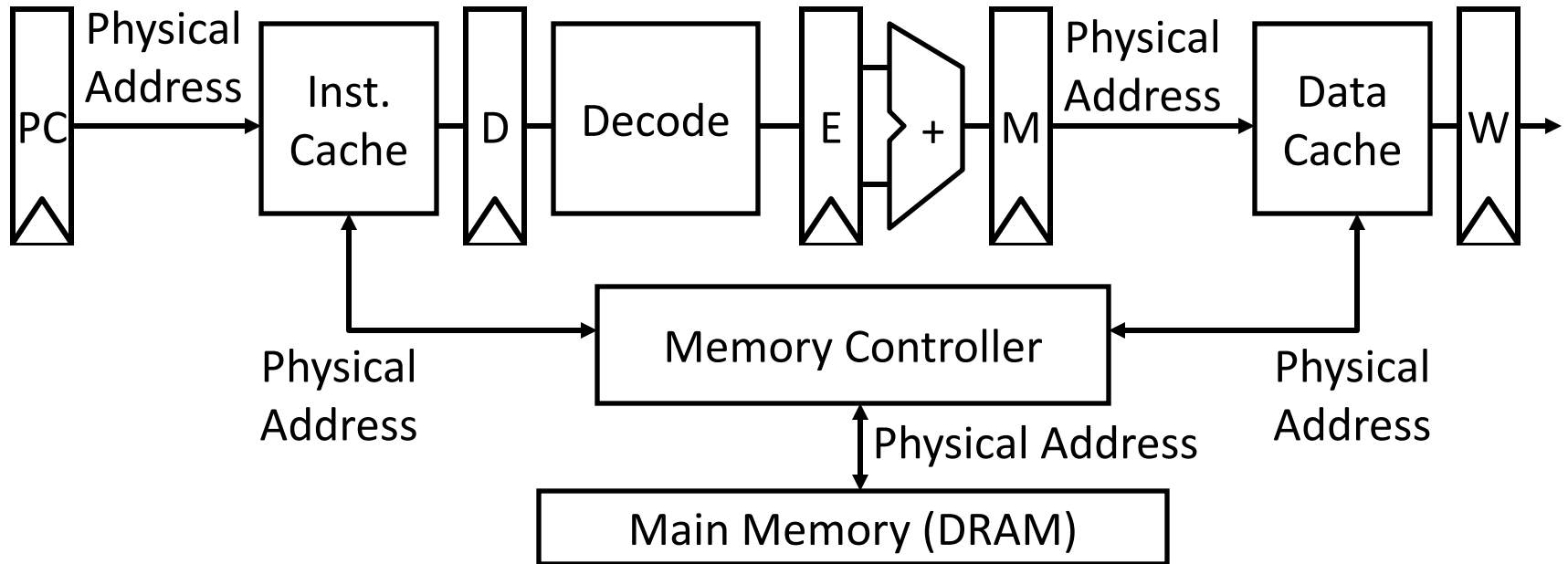
CENG331 - Computer Organization

**Instructor:**

Murat Manguoglu (Section 1)

Adapted from: <http://inst.eecs.berkeley.edu/~cs152>

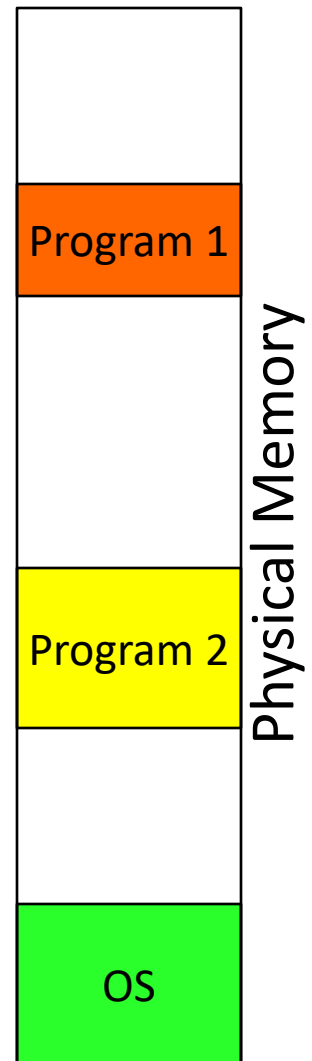
# Bare Machine



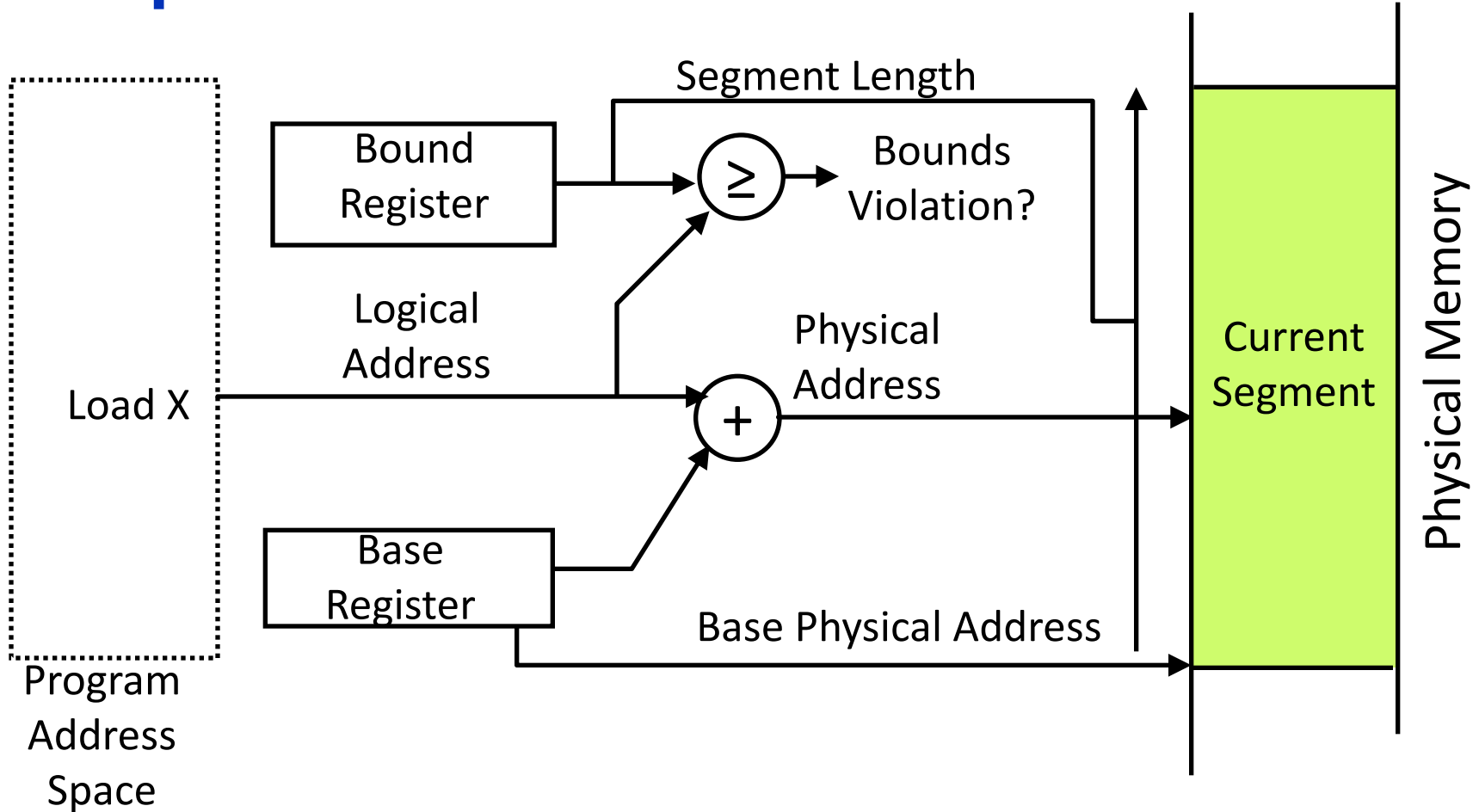
In a bare machine, the only kind of address is a physical address, corresponding to address lines of actual hardware memory.

# Dynamic Address Translation

- Motivation
  - In early machines, I/O was slow and each I/O transfer involved the CPU (programmed I/O)
  - Higher throughput possible if CPU and I/O of 2 or more programs were overlapped, how?
    - multiprogramming with DMA I/O devices, interrupts
- 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 software to manage context switches between multiple programs



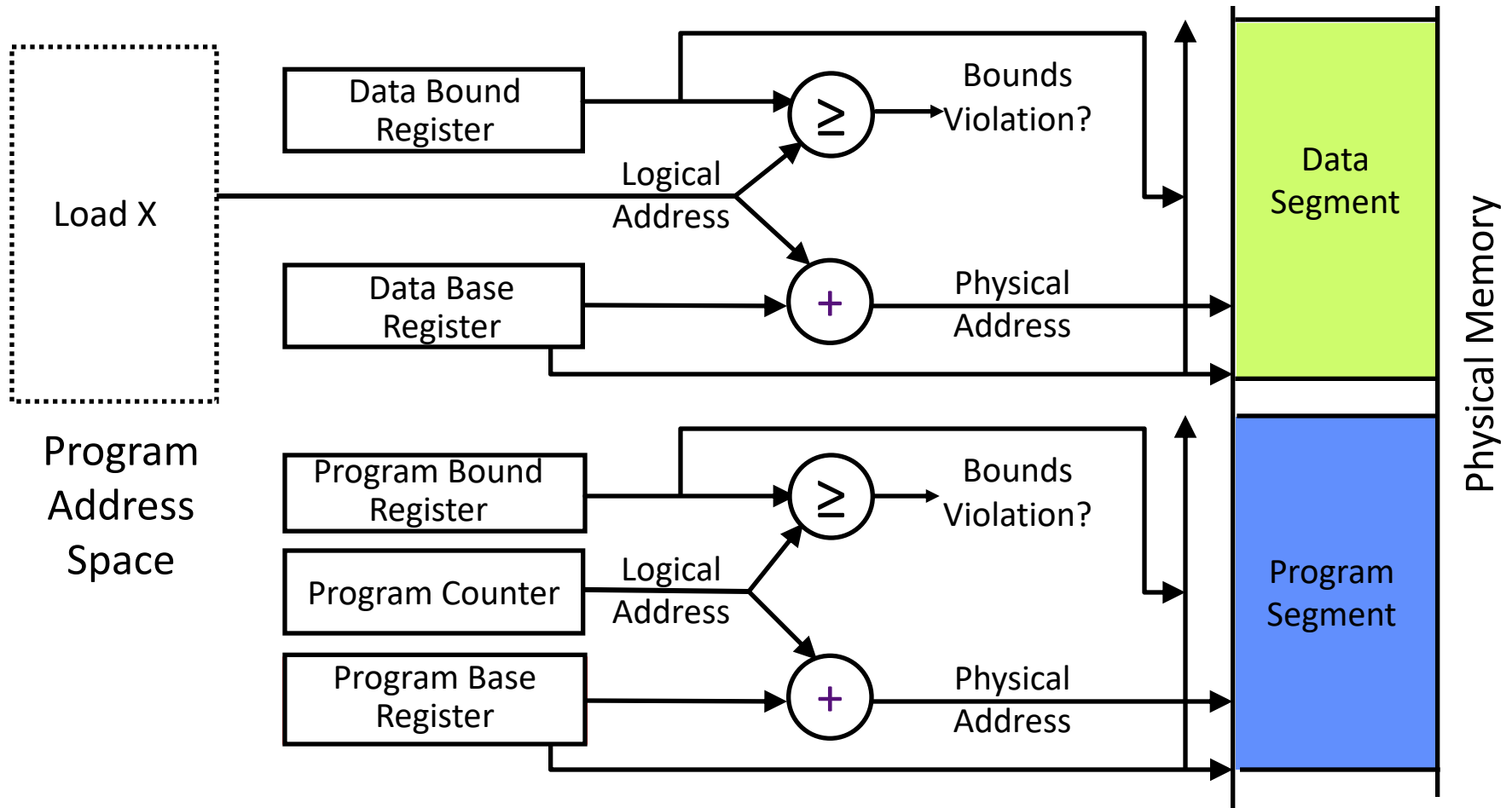
# Simple Base and Bound Translation



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

# Separate Areas for Program and Data

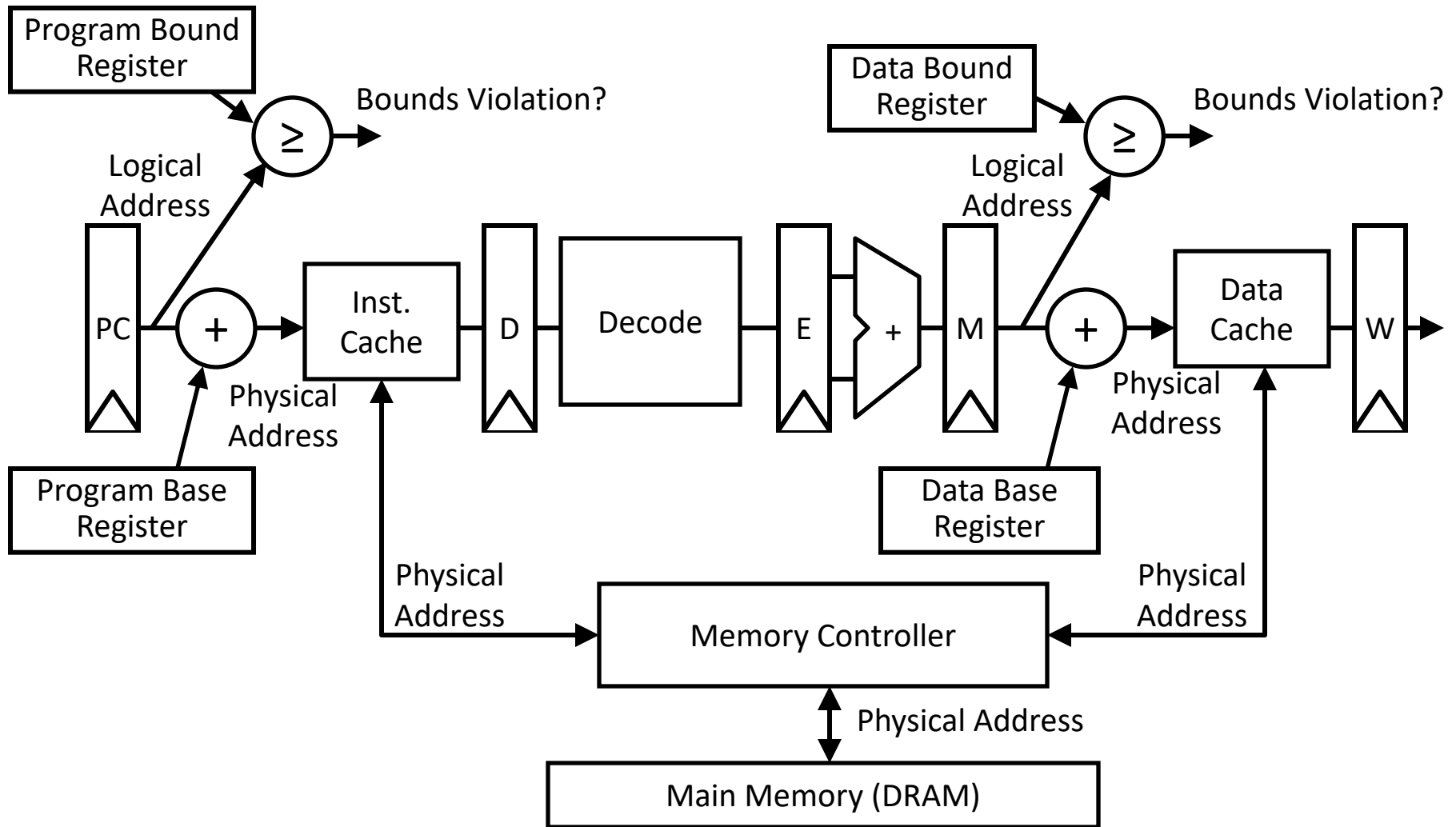
(Scheme used on all Cray vector supercomputers prior to X1, 2002)



*What is an advantage of this separation?*

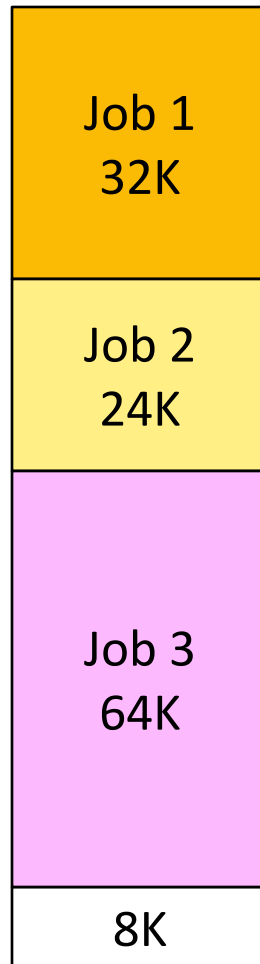
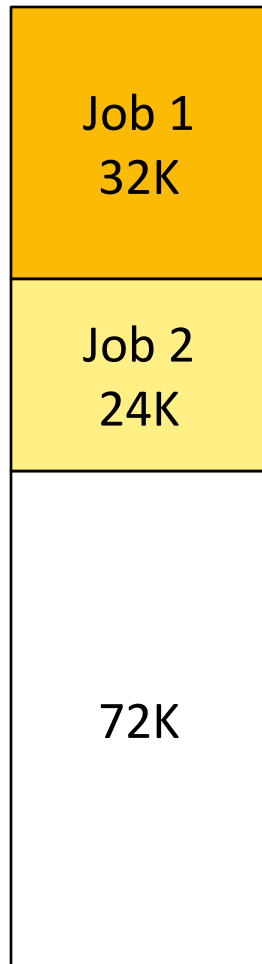
*What about more base/bound pairs?*

# Base and Bound Machine

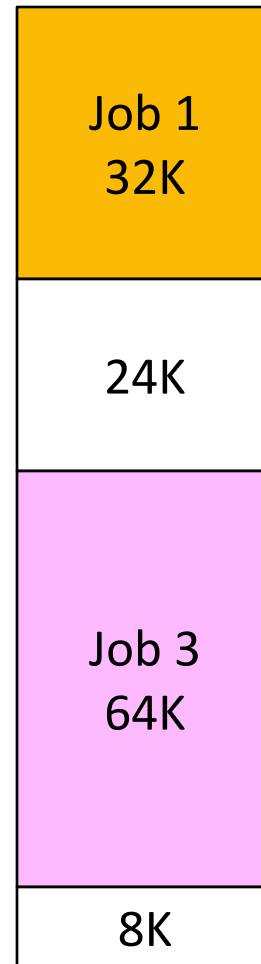


*Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)*

# External Fragmentation with Segments

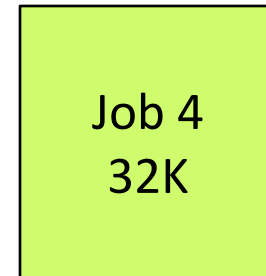


Job 3  
starts



Job 2  
finishes

*Can't run Job 4, as  
not enough  
contiguous space.  
Must compact.*



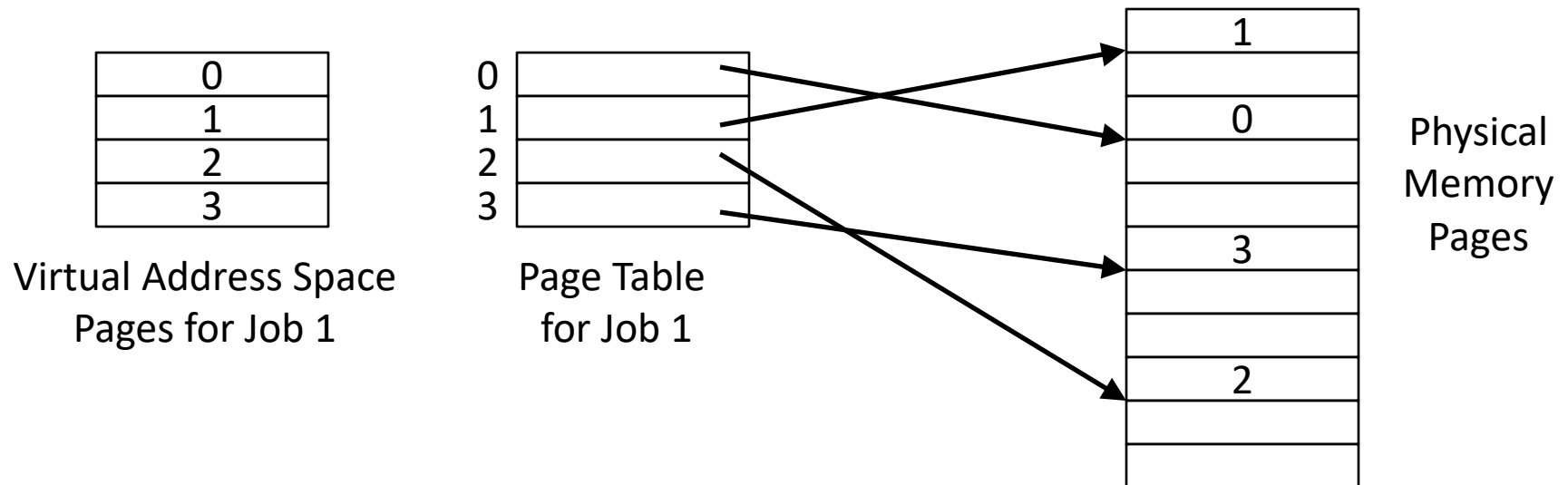
Job 4  
arrives

# Paged Memory Systems

- Program-generated (*virtual* or *logical*) address split into:

Page Number	Offset
-------------	--------

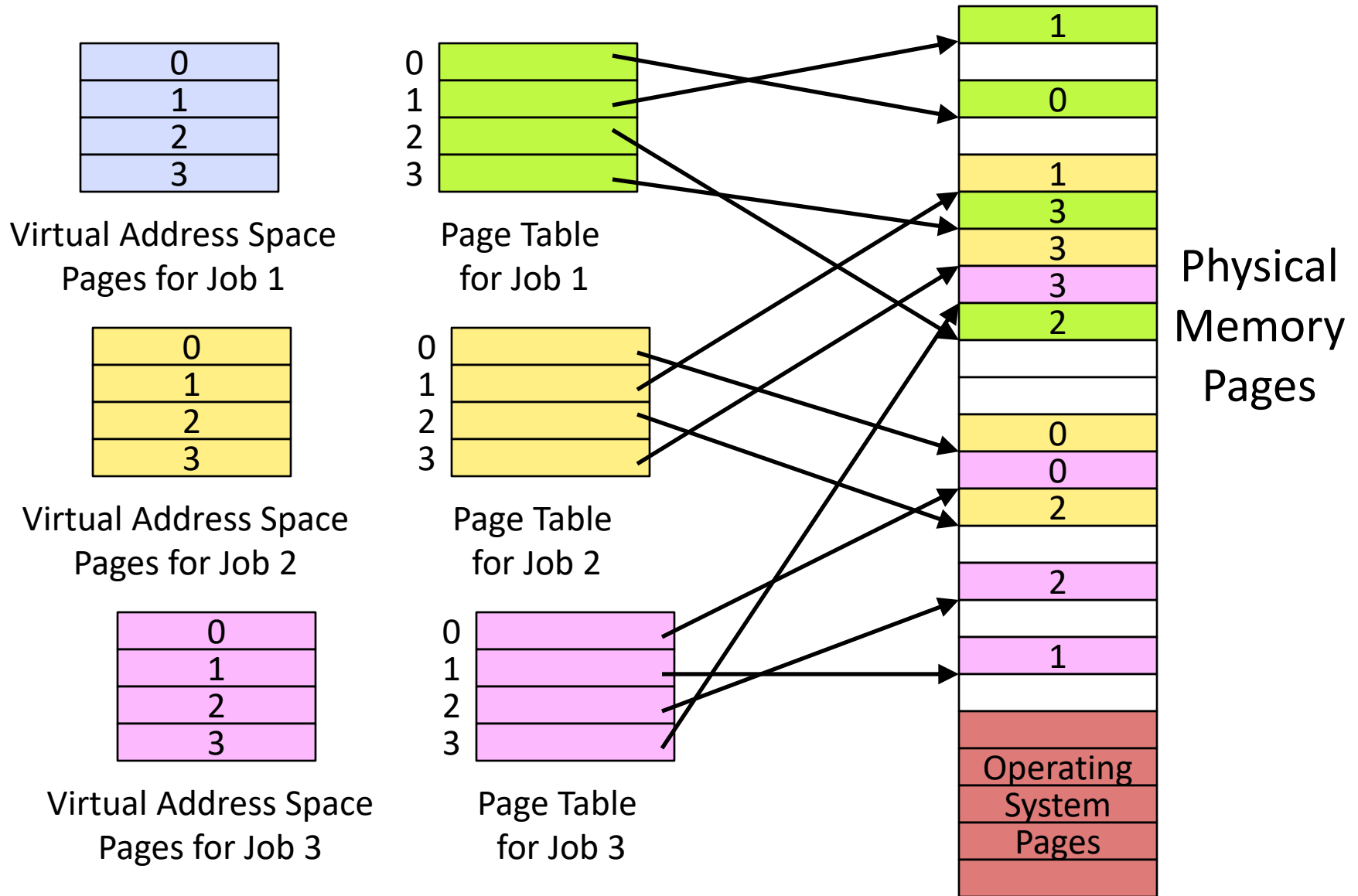
- Page Table contains physical address of start of each fixed-sized page in virtual address space



- Paging makes it possible to store a large contiguous virtual memory space using non-contiguous physical memory pages



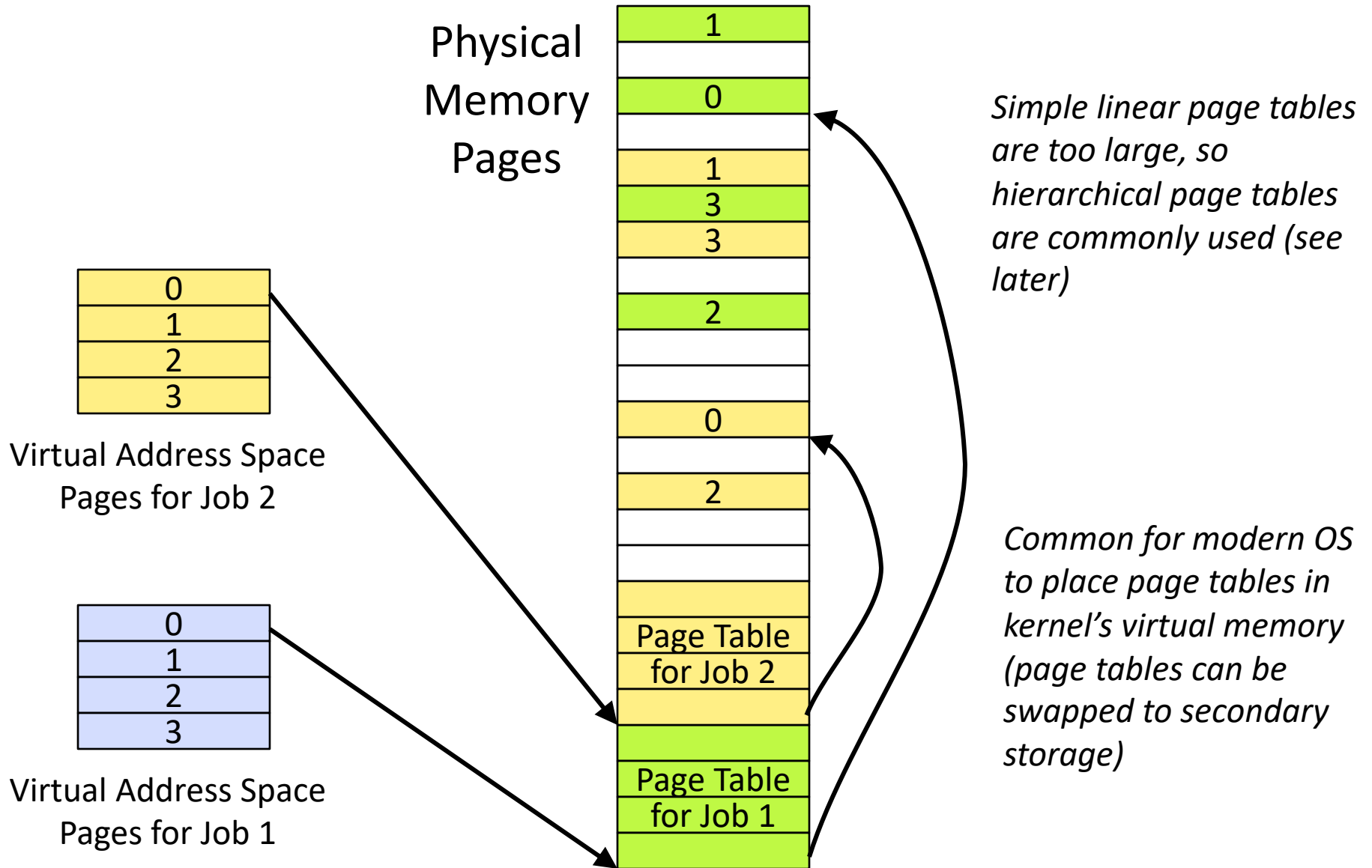
# Private Address Space per User



# Paging Simplifies Allocation

- Fixed-size pages can be kept on OS free list and allocated as needed to any process
- Process memory usage can easily grow and shrink dynamically
- Paging suffers from *internal fragmentation* where not all bytes on a page are used
  - Much less of an issue than external fragmentation or compaction for common page sizes (4-8KB)
  - But one reason that many oppose move to larger page sizes

# Page Tables Live in Memory



# Coping with Limited Primary Storage

- Paging reduces fragmentation, but still many problems would not fit into primary memory, have to copy data to and from secondary storage (drum, disk)
- Two early approaches:
  - **Manual overlays**, programmer explicitly copies code and data in and out of primary memory
    - Tedious coding, error-prone (jumping to non-resident code?)
  - **Software interpretive coding** (Brooker 1960). Dynamic interpreter detects variables that are swapped out to drum and brings them back in
    - Simple for programmer, but inefficient

*Not just ancient black art, e.g., IBM Cell microprocessor using in Playstation-3 had explicitly managed local store!*

*Many new “deep learning” accelerators have similar structure.*

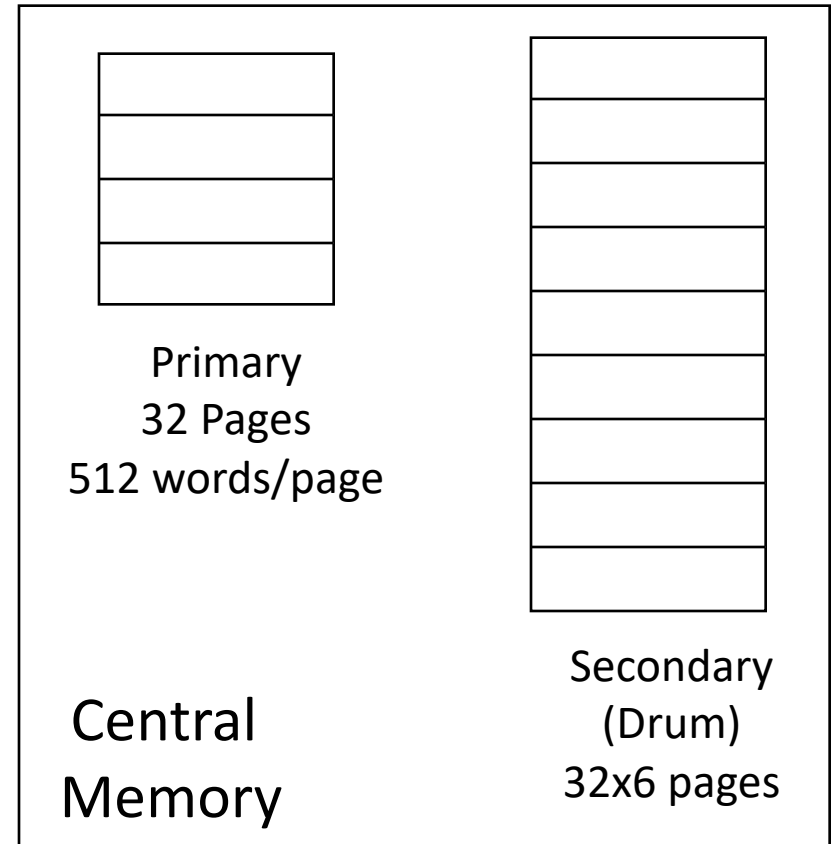
# Demand Paging in Atlas (1962)

“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

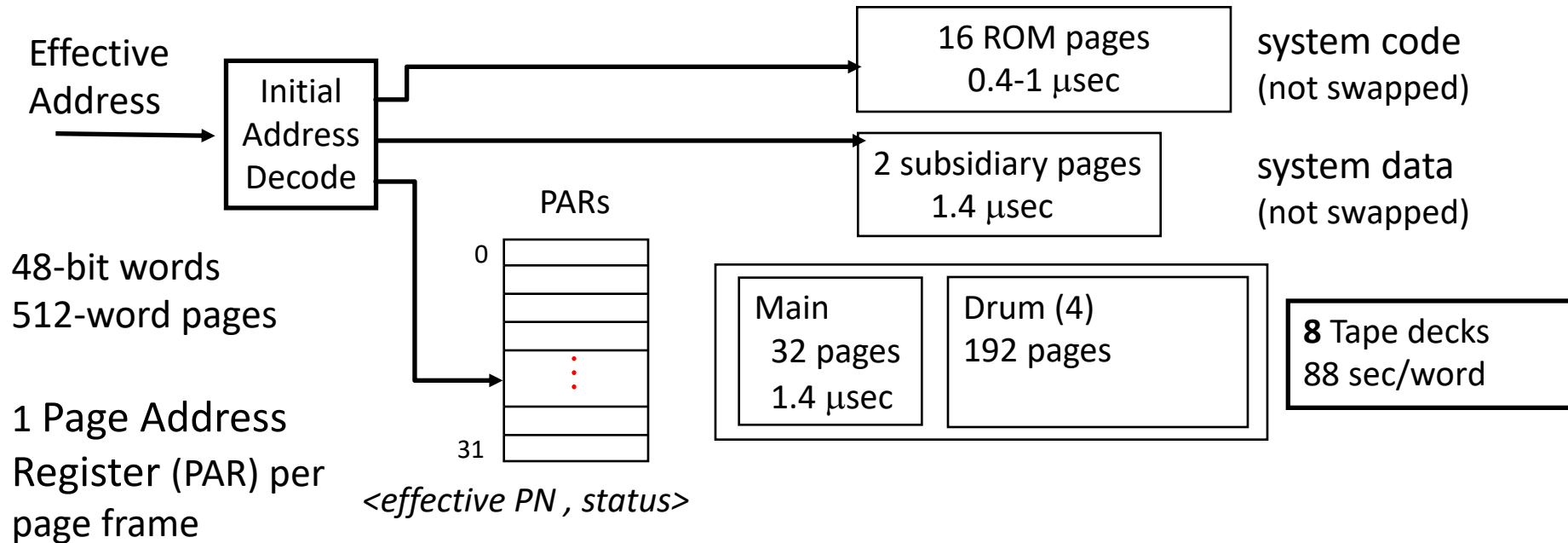
*Tom Kilburn*

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



# Hardware Organization of Atlas



Compare the effective page address against all 32 PARs

match  $\Rightarrow$  normal access

no match  $\Rightarrow$  *page fault*

save the state of the partially executed instruction

# Atlas Demand-Paging Scheme

On a page fault:

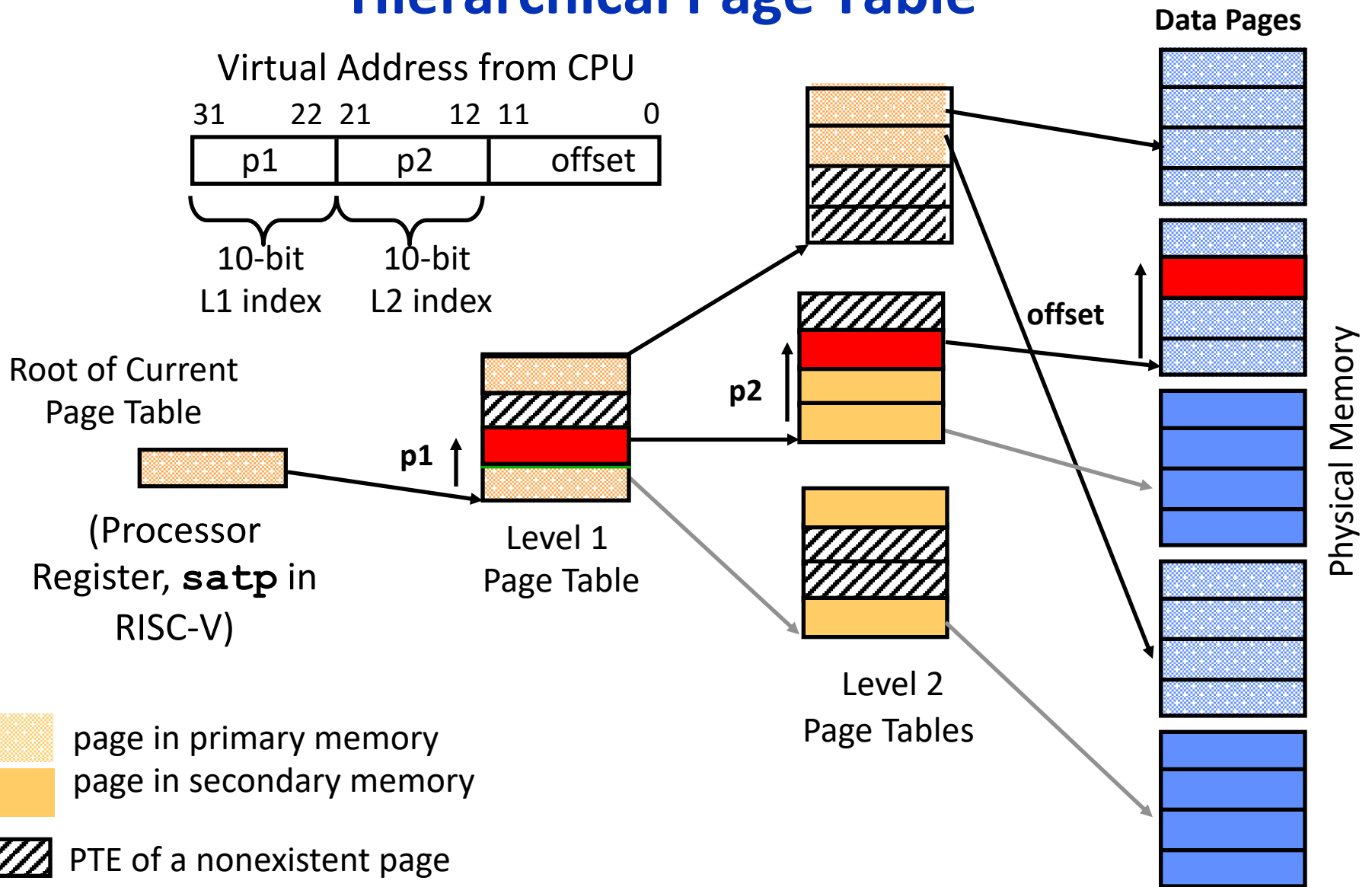
- Input transfer into a free page is initiated
- The Page Address Register (PAR) is updated
- If no free page is left, a page is selected to be replaced (based on usage)
- The replaced page is written on the drum
  - to minimize drum latency effect, the first empty page on the drum was selected
- The page table is updated to point to the new location of the page on the drum

# Size of Linear Page Table

- With 32-bit addresses, 4-KB pages & 4-byte PTEs:
  - $2^{32}/2^{12} = 2^{20}$  virtual pages per user, assuming 4 Byte PTEs
  - $2^{20}$  PTEs, i.e, 4 MB page table per user
  - 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!)

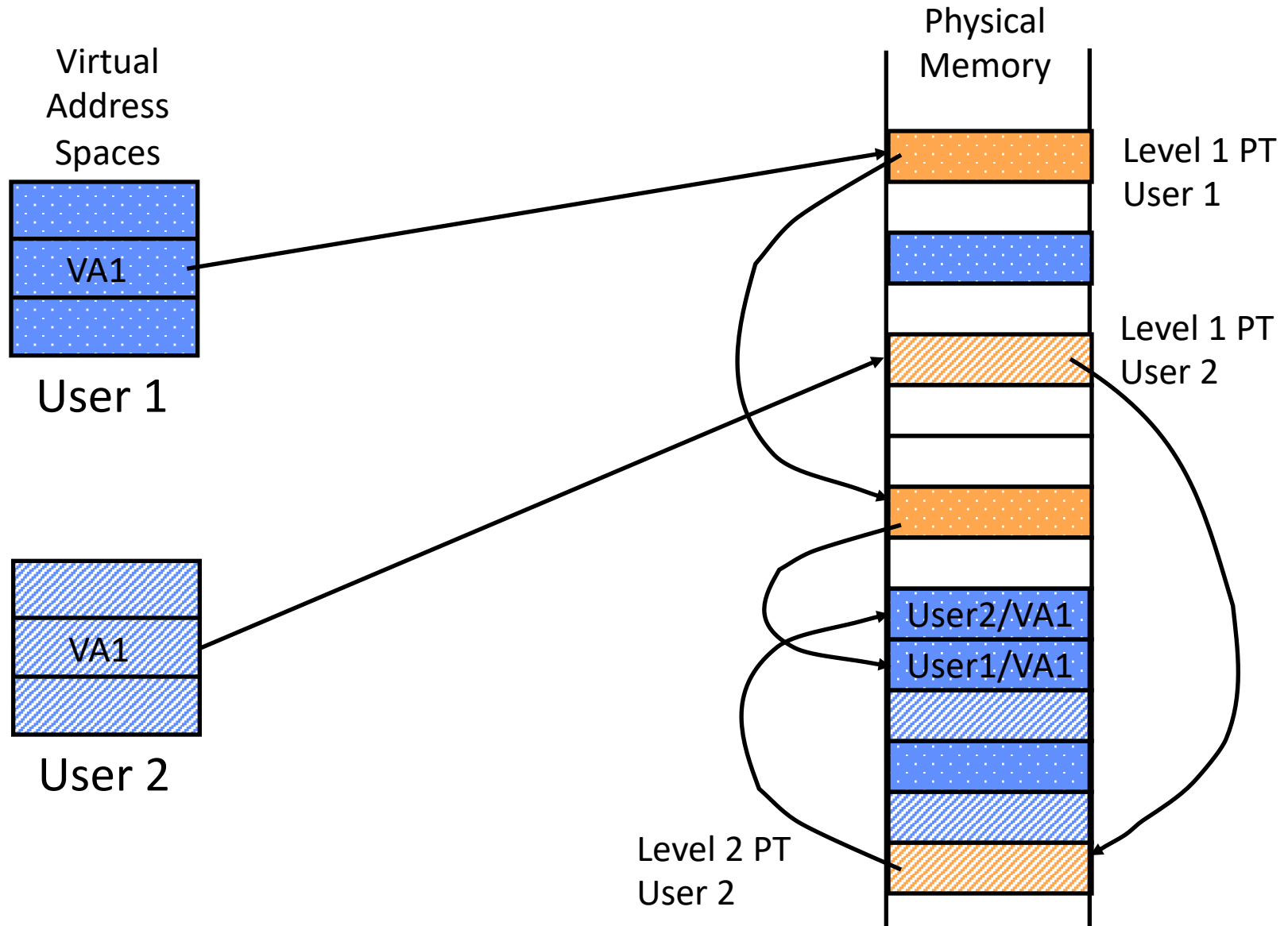


# Hierarchical Page Table

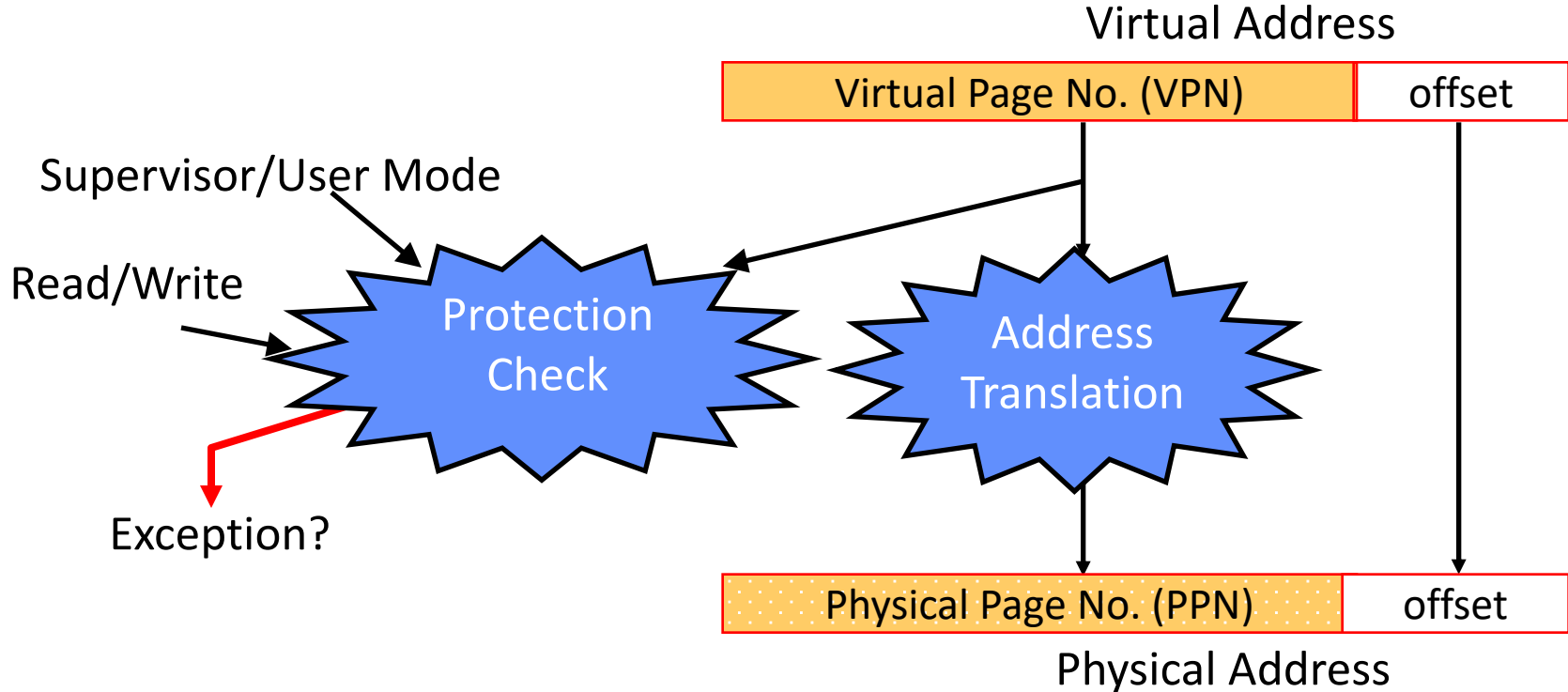


RISC-V Sv32 Virtual Memory Scheme

# Two-Level Page Tables in Physical Memory



# Address Translation & Protection



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

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

# Translation-Lookaside Buffers (TLB)

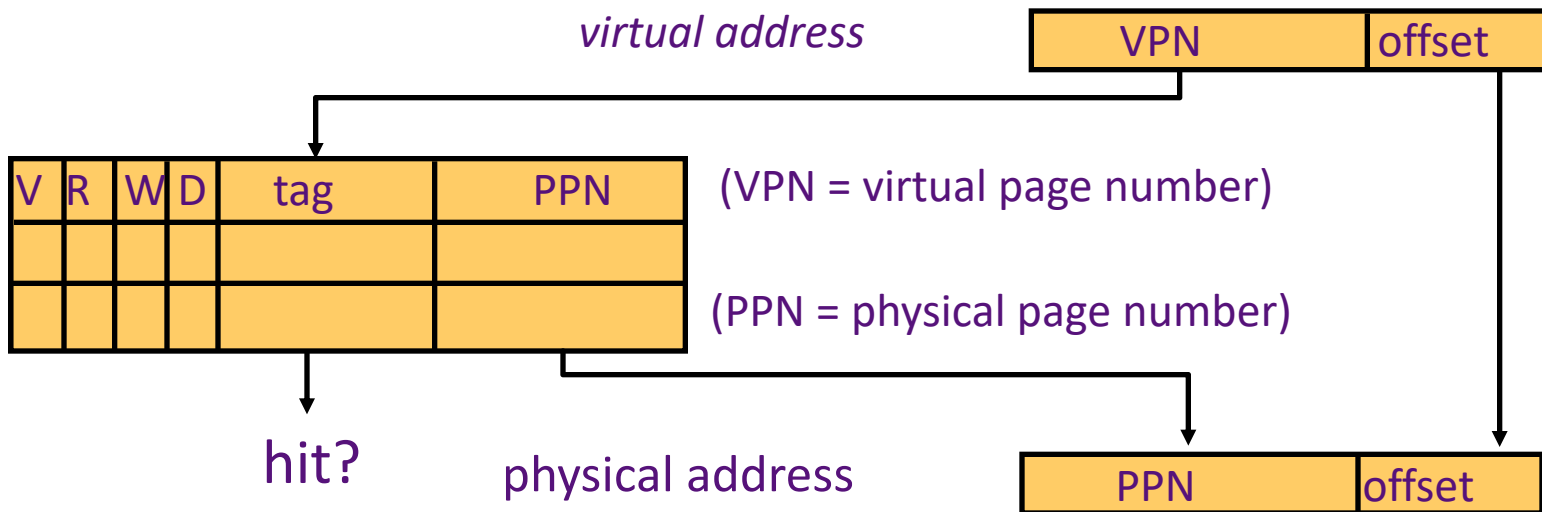
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit  $\Rightarrow$  *Single-Cycle Translation*

TLB miss  $\Rightarrow$  *Page-Table Walk to refill*



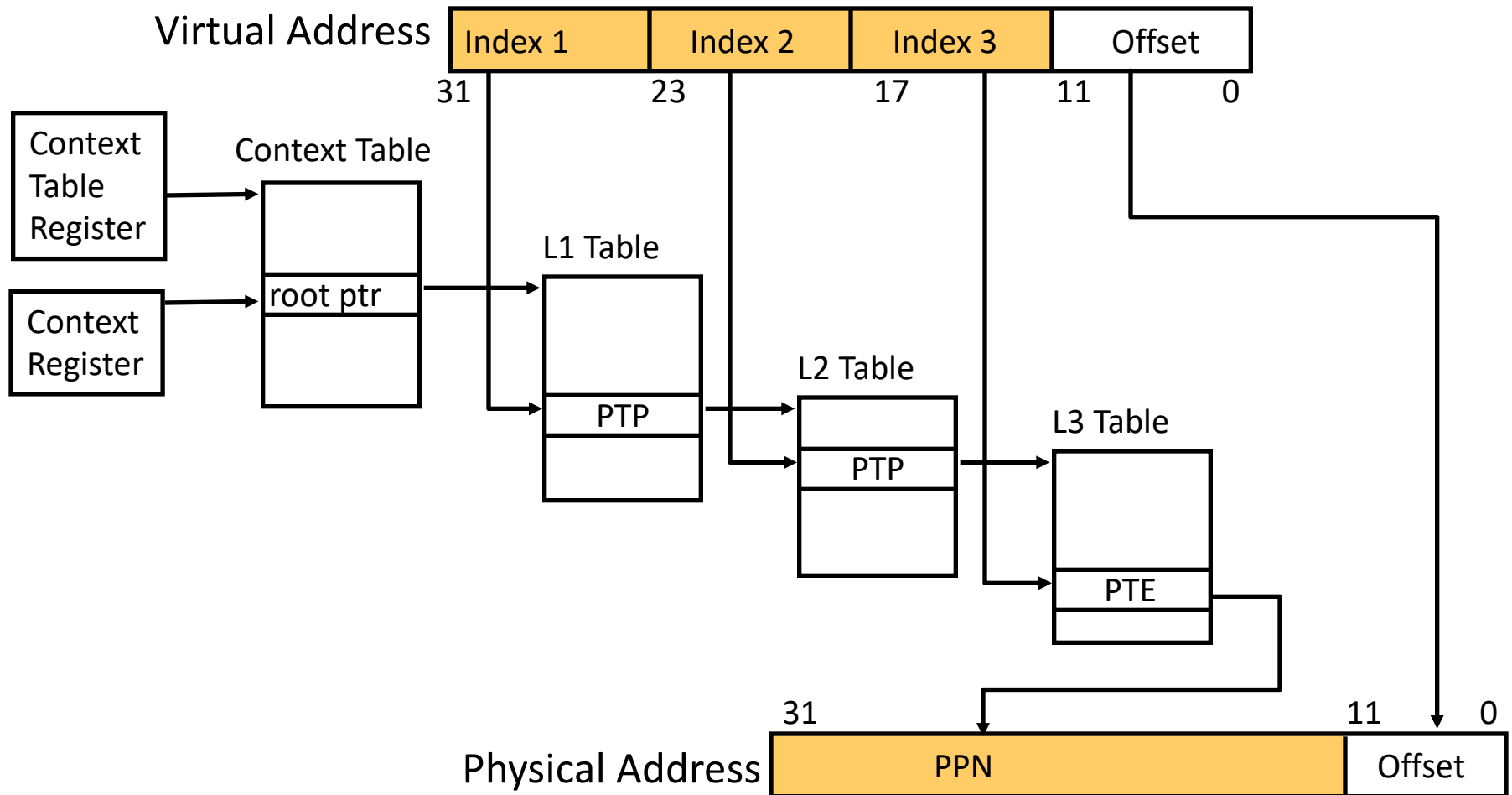
# TLB Designs

- Typically 32-128 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 or FIFO replacement policy
- 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 =  $64 \text{ entries} * 4 \text{ KB} = 256 \text{ KB (if contiguous)}$

# 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.
  - Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.
- Hardware (*SPARC v8, x86, PowerPC, RISC-V*)
  - 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.
- NOTE: A given ISA can use either TLB miss strategy

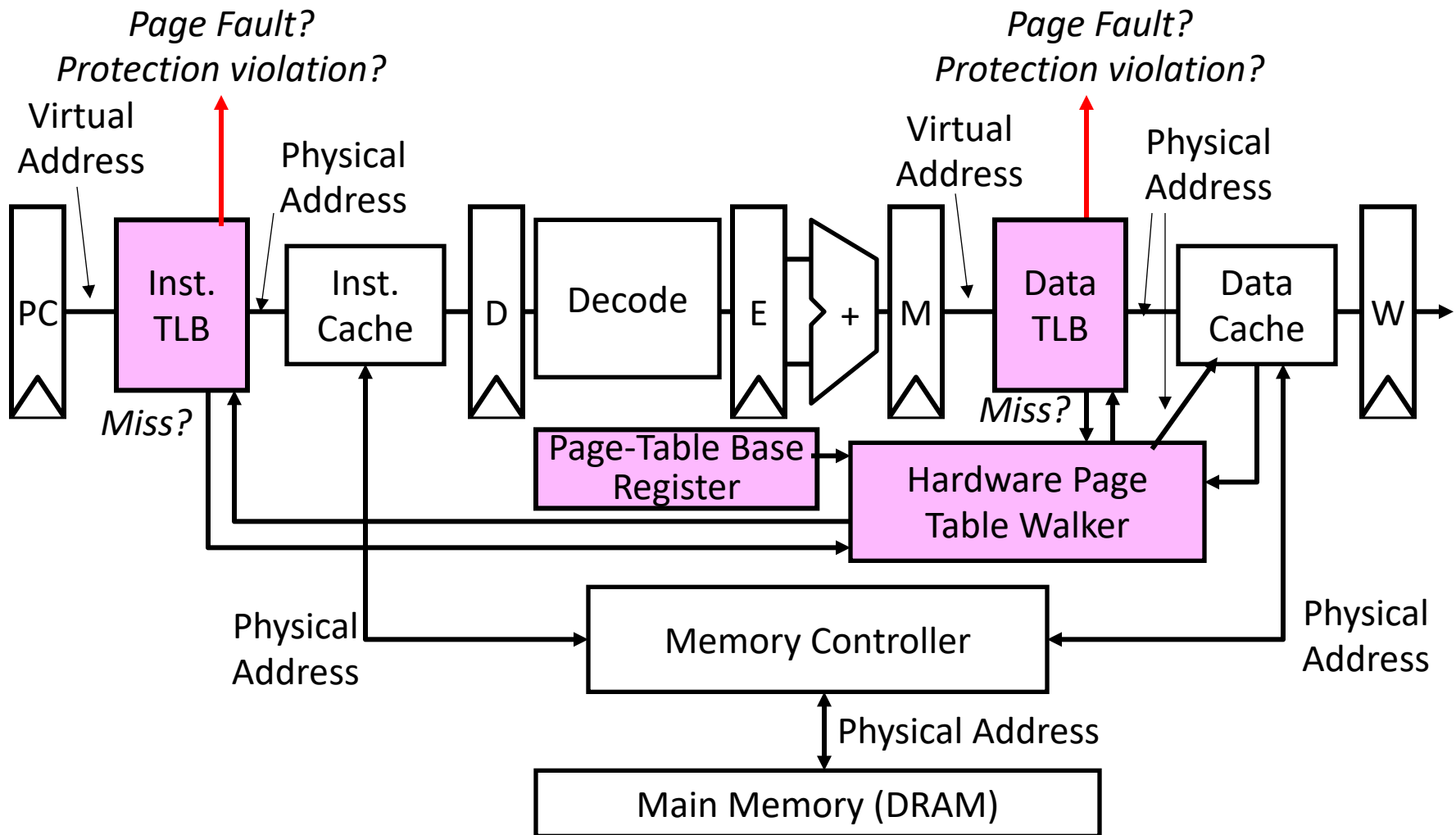
# Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

# Page-Based Virtual-Memory Machine

## (Hardware Page-Table Walk)



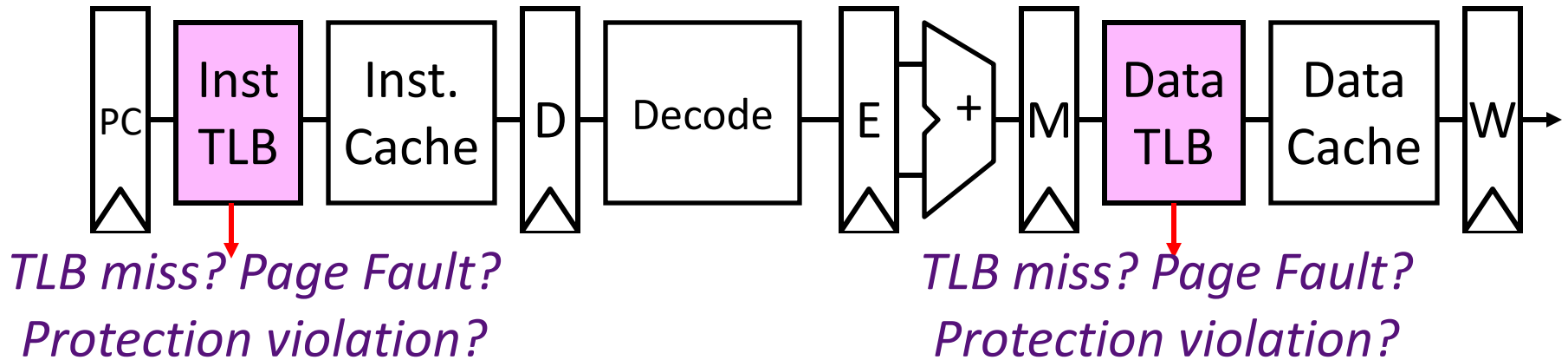
- Assumes page tables held in untranslated physical memory



# Page-Fault Handler

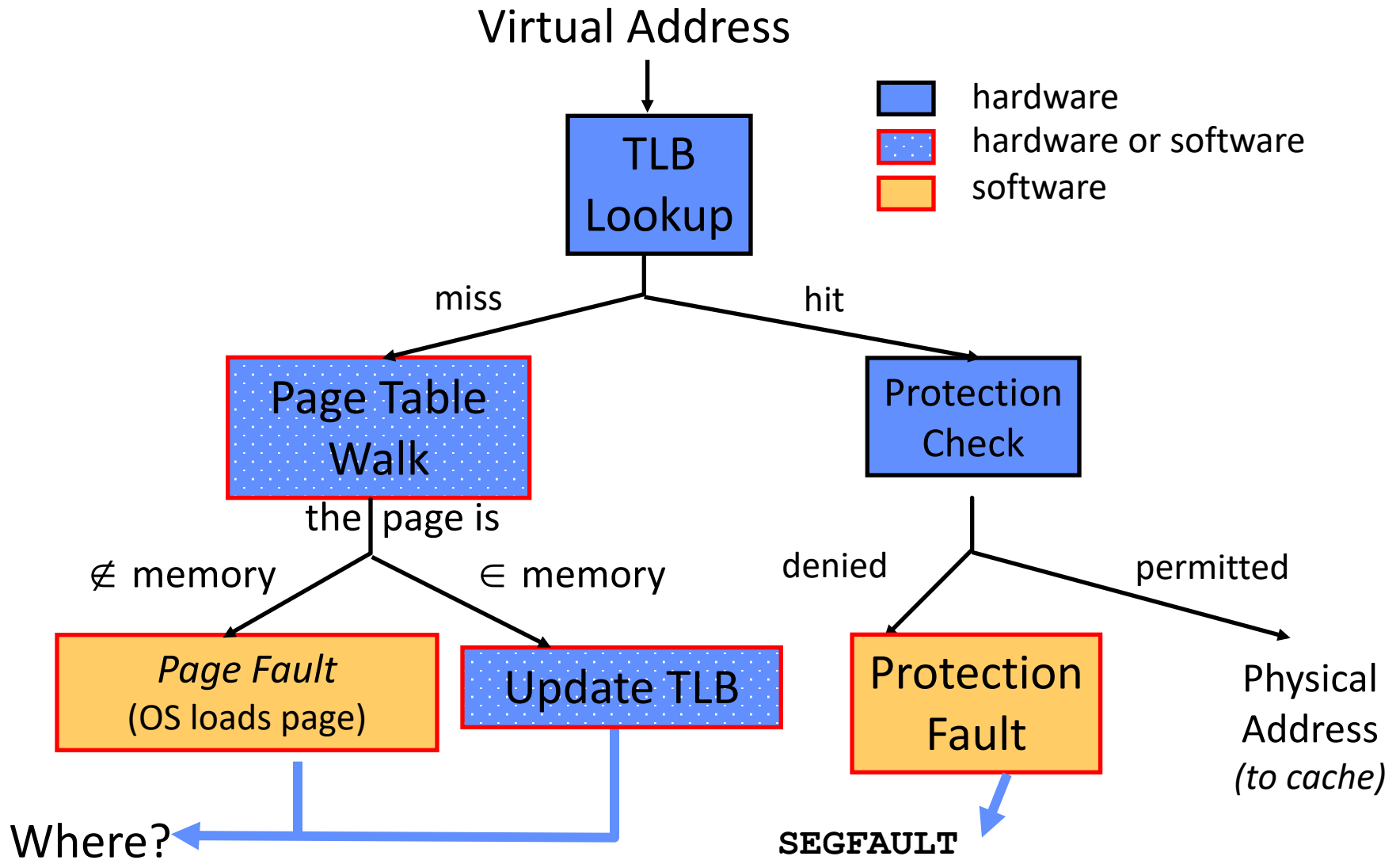
- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped 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 OS
  - Untranslated addressing mode is essential to allow kernel to access page tables
- Keeping TLBs coherent with page table changes might require expensive “TLB shutdown”
  - Interrupt other processors to invalidate stale TLB entries
  - Some mainframes had hardware TLB coherence

# Handling VM-related exceptions

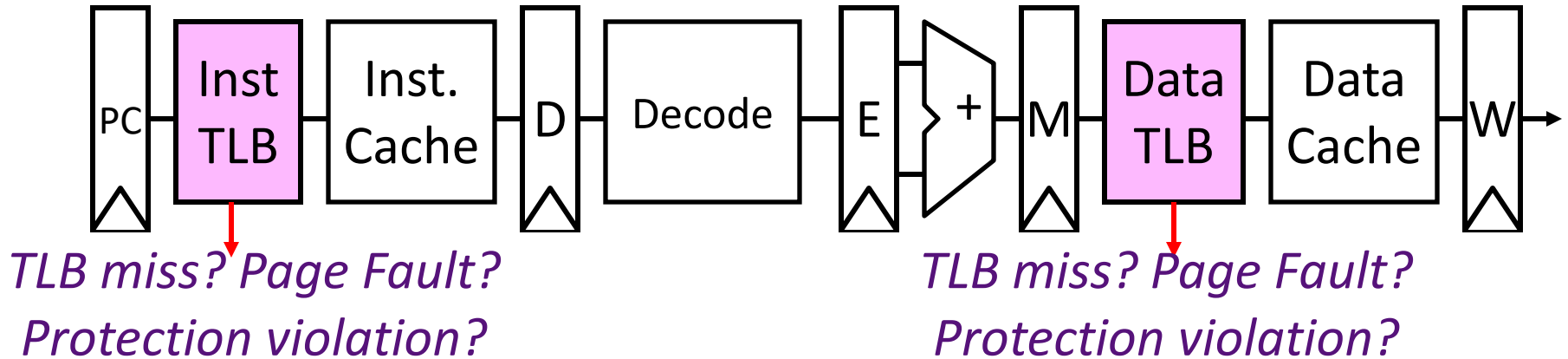


- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Handling page fault (e.g., page is on disk) needs *restartable* exception so software handler can resume after retrieving page
  - Precise exceptions are easy to restart
  - Can be imprecise but restartable, but this complicates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

# Address Translation: *putting it all together*

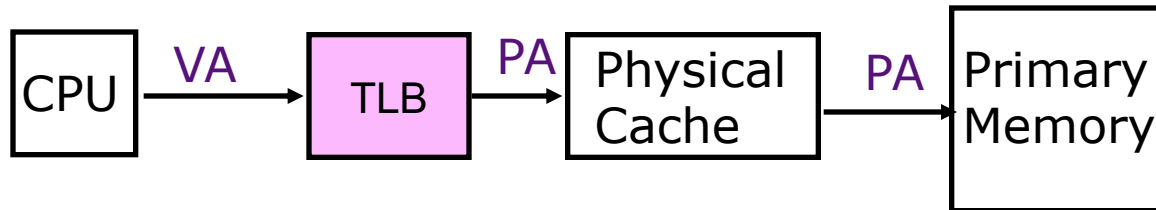


# Address Translation in CPU Pipeline

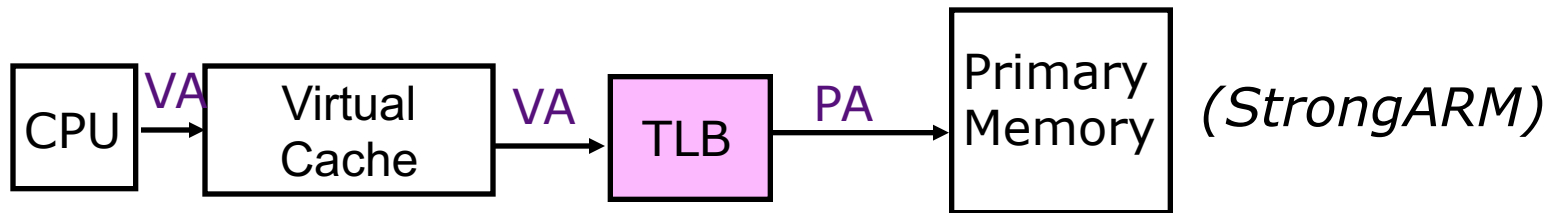


- Need to cope with additional latency of TLB:
  - slow down the clock?
  - pipeline the TLB and cache access?
  - virtual address caches
  - parallel TLB/cache access

# Virtual-Address Caches

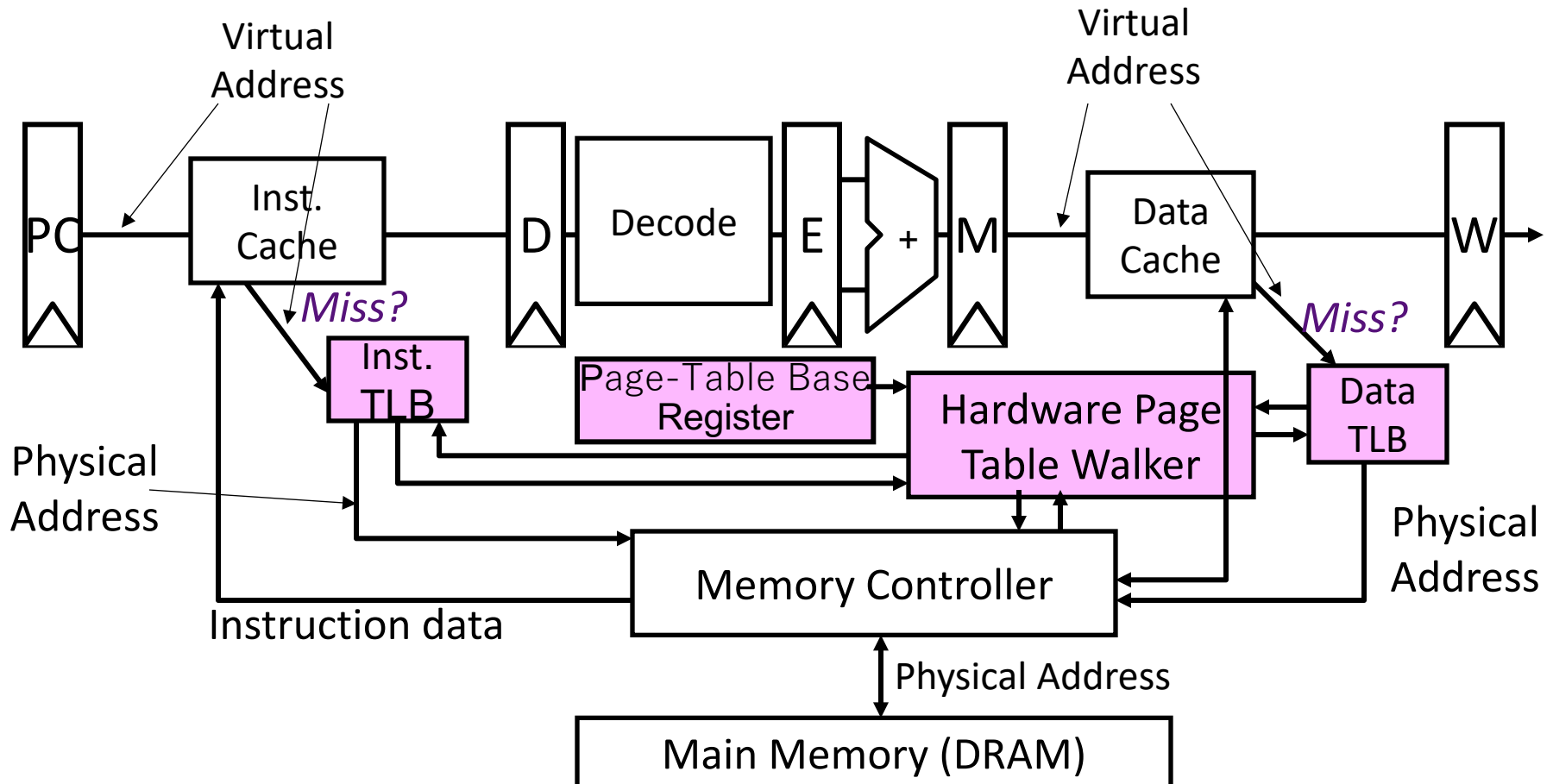


*Alternative: place the cache before the TLB*



- 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*

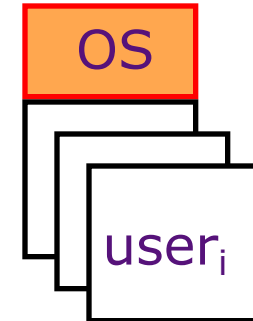
# Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

## Protection & Privacy

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

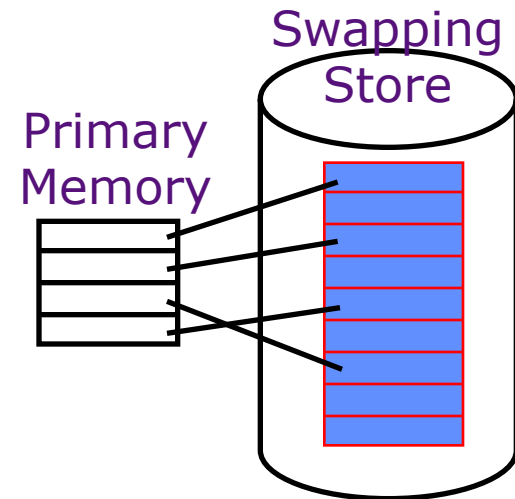
page table  $\equiv$  name space



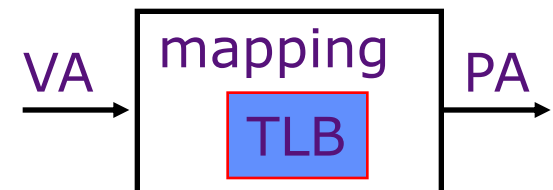
## Demand Paging

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*



# 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
- Modern GPUs operate similarly to vector supercomputers, with translation and protection but not demand paging

## 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 precise or restartable exceptions for exposed architectures

# 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)
- MIT material derived from course 6.823
- UCB material derived from course CS252