CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 8 – Address Translation

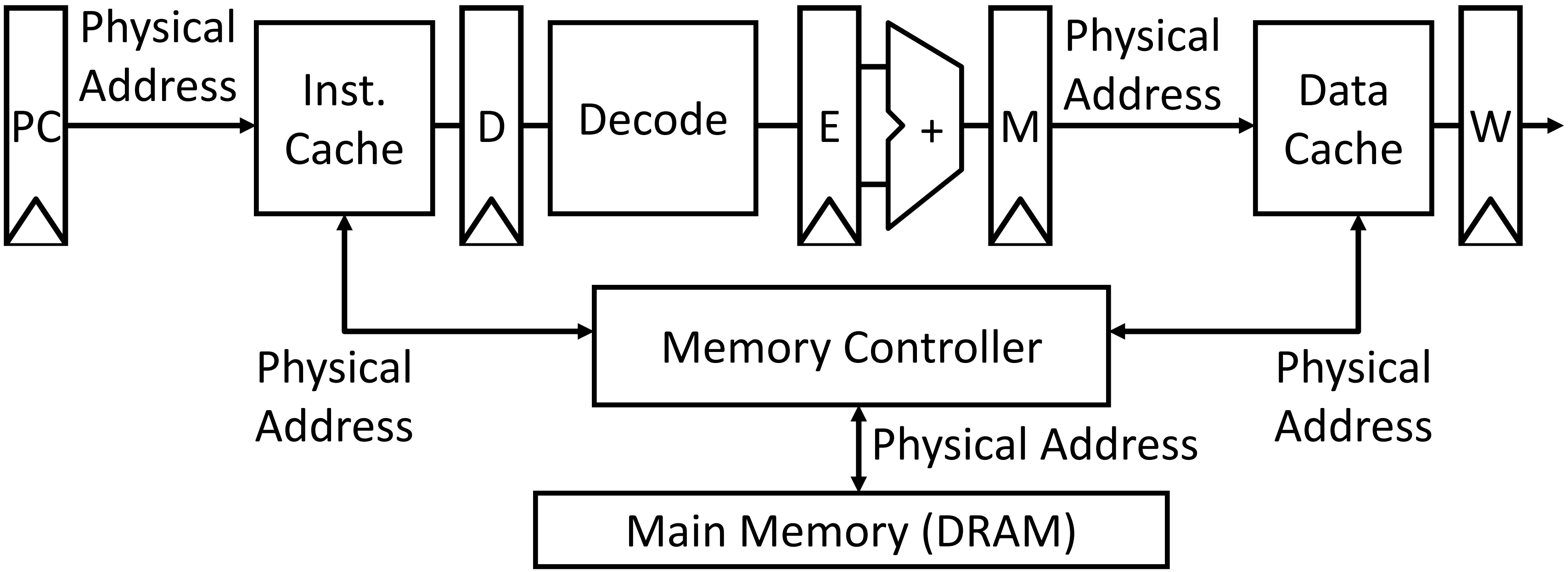
Chris Fletcher
Electrical Engineering and Computer Sciences
University of California at Berkeley

https://cwfletcher.github.io/
http://inst.eecs.berkeley.edu/~cs152

Last time in Lecture 7

- Prefetching, hardware or software
  - correctness, timeliness
  - instructions easier to prefetch than data
  - software difficult to use ideally

Bare Machine



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

Managing Memory in Bare Machines

- Early machines only ran one program at a time, with this program having unrestricted access to all memory and all I/O devices
  - This simple memory management model was also used in turn by the first minicomputer and first microcomputer systems
- Subroutine libraries became popular, were written in location-independent form
  - Different programs use different combination of routines
- To run program on bare machines, use *linker* or *loader* program to relocate library modules to actual locations in physical memory

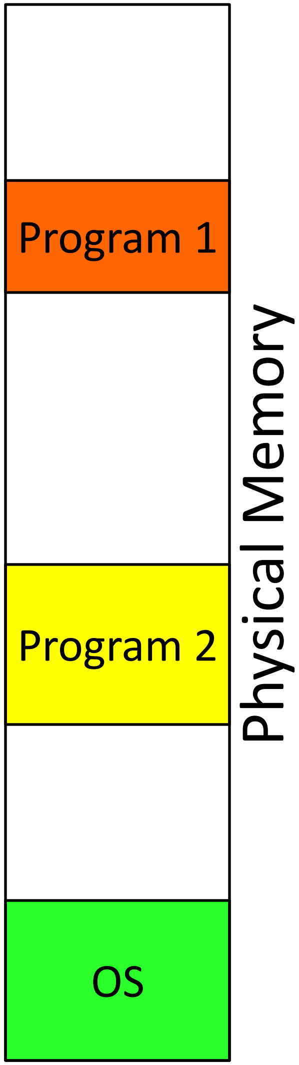
Dynamic Address Translation

- 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?

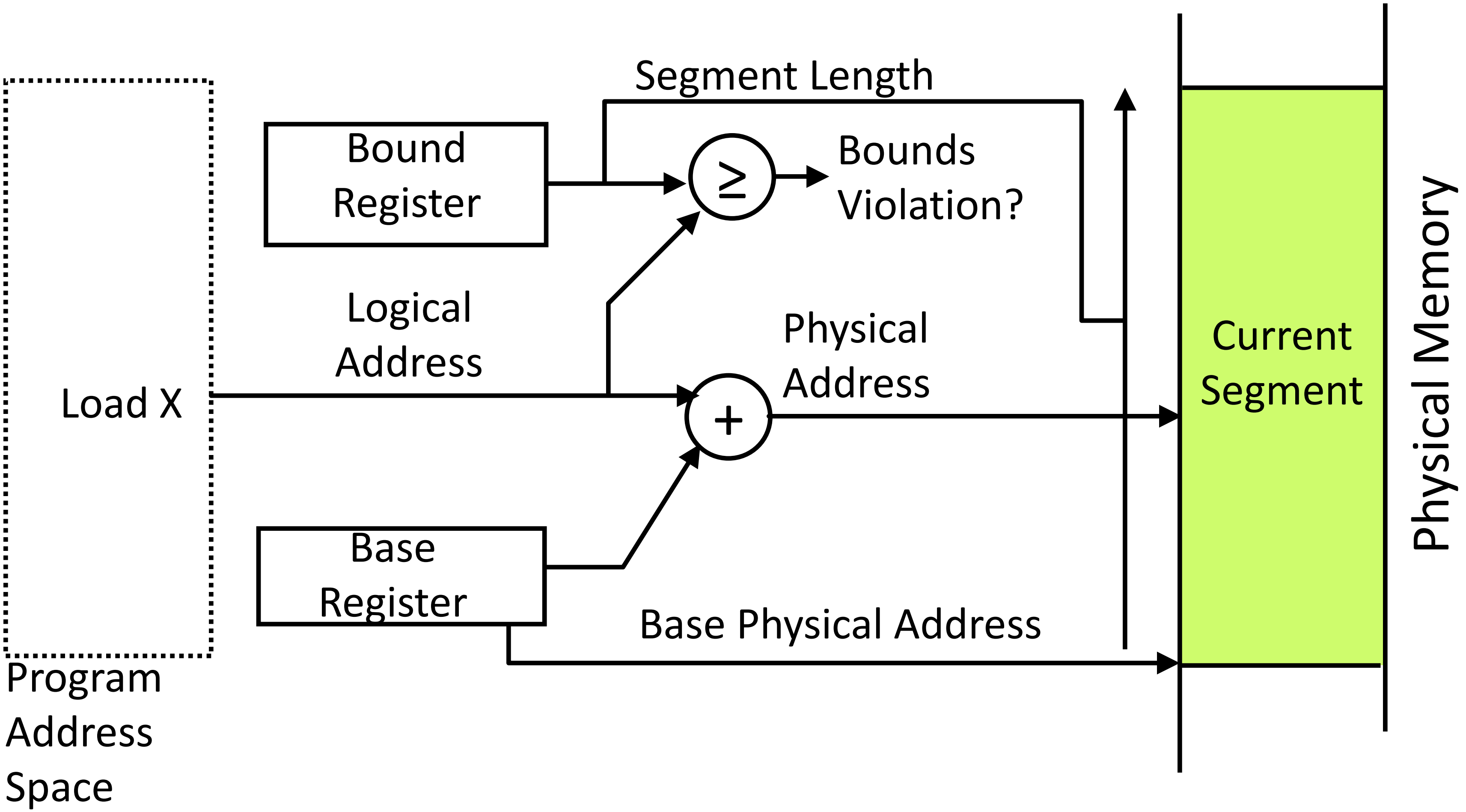
  → multiprogramming with DMA I/O devices, interrupts
- Location-independent programs
  - Programming and storage management ease

  → 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



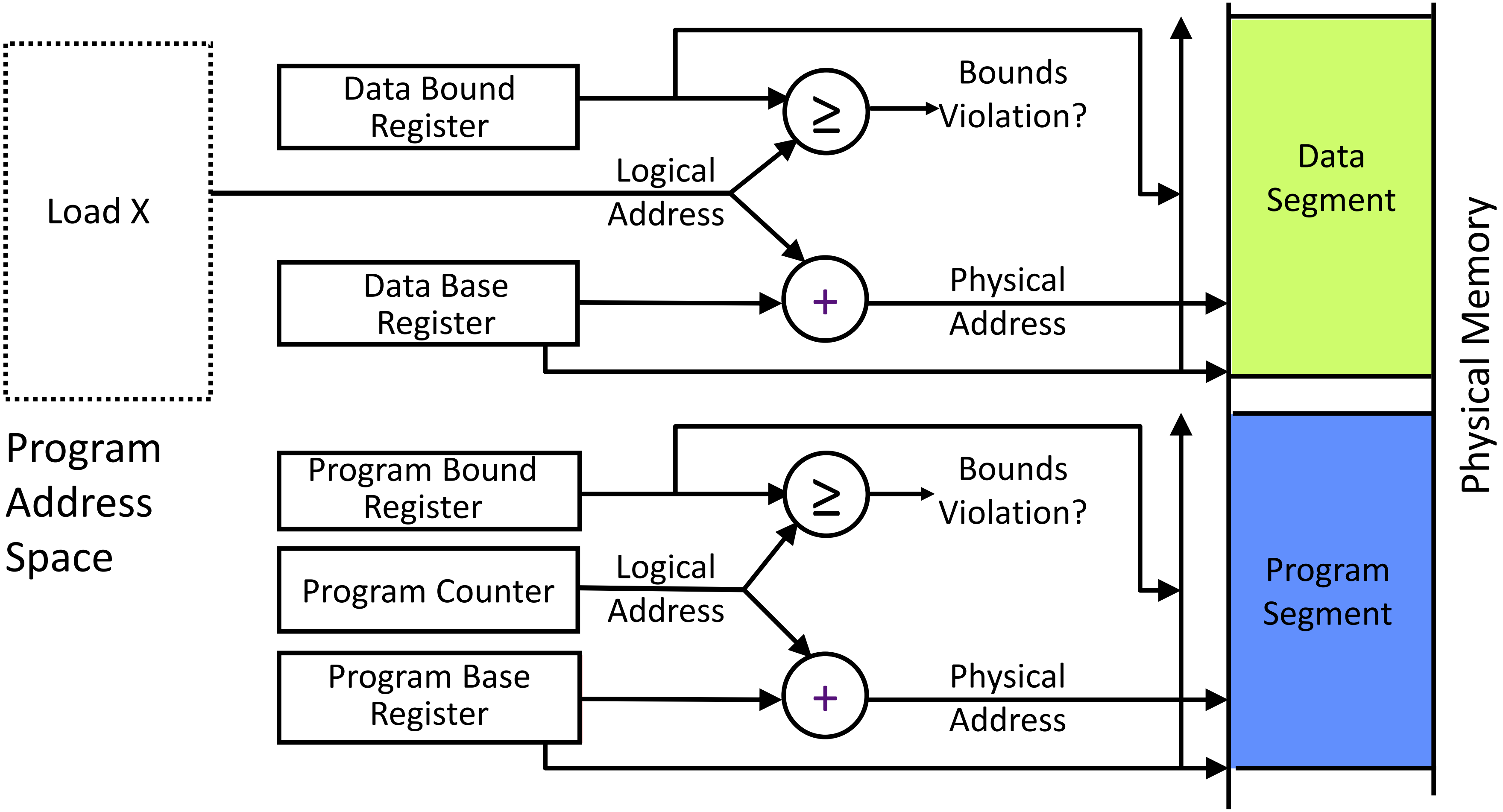
Simple Base and Bound Translation



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

Separate Areas for Program and Data

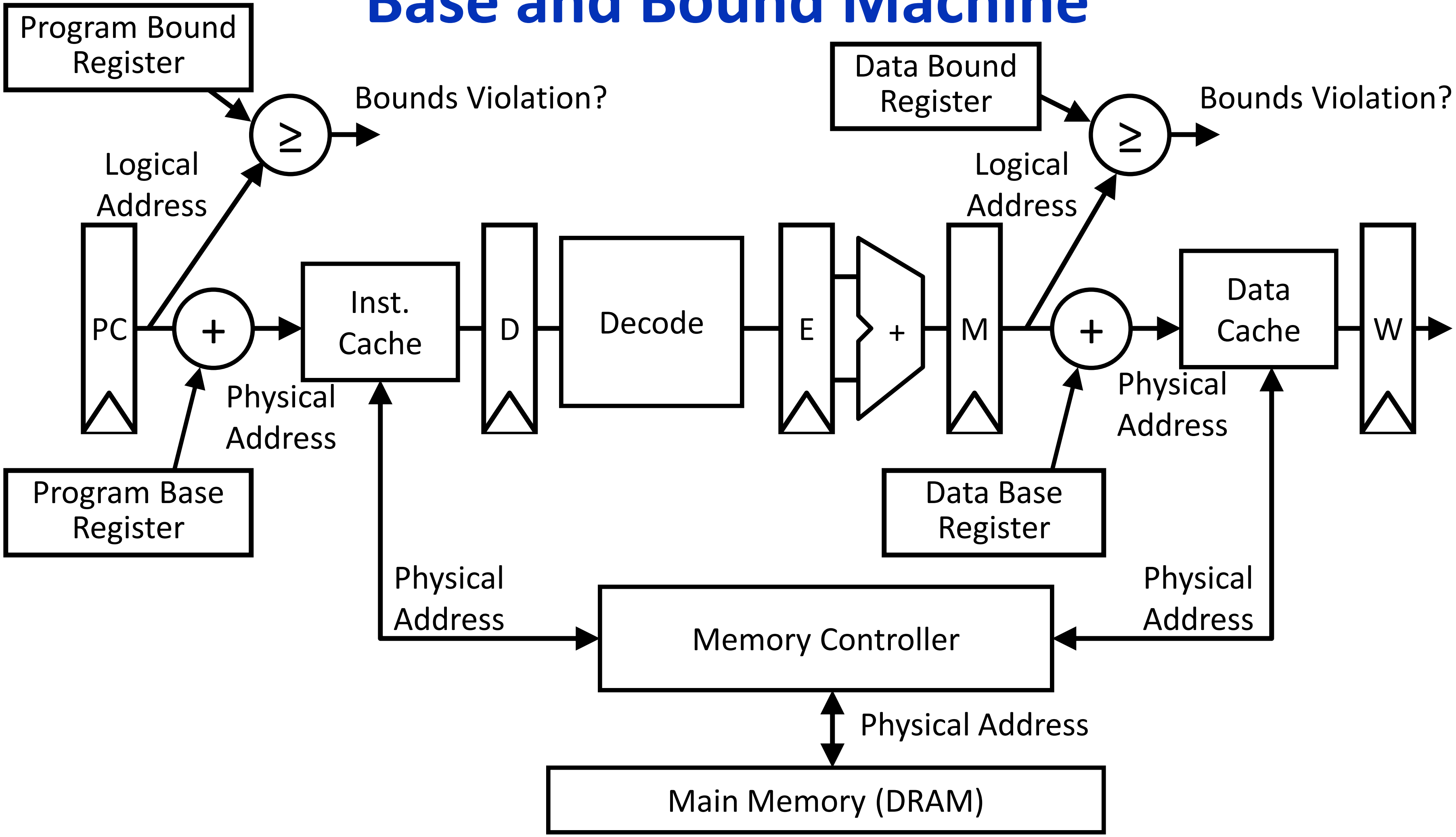
(Scheme used on all Cray vector supercomputers prior to X1, 2002)



What is an advantage of this separation?

What about more base/bound pairs?

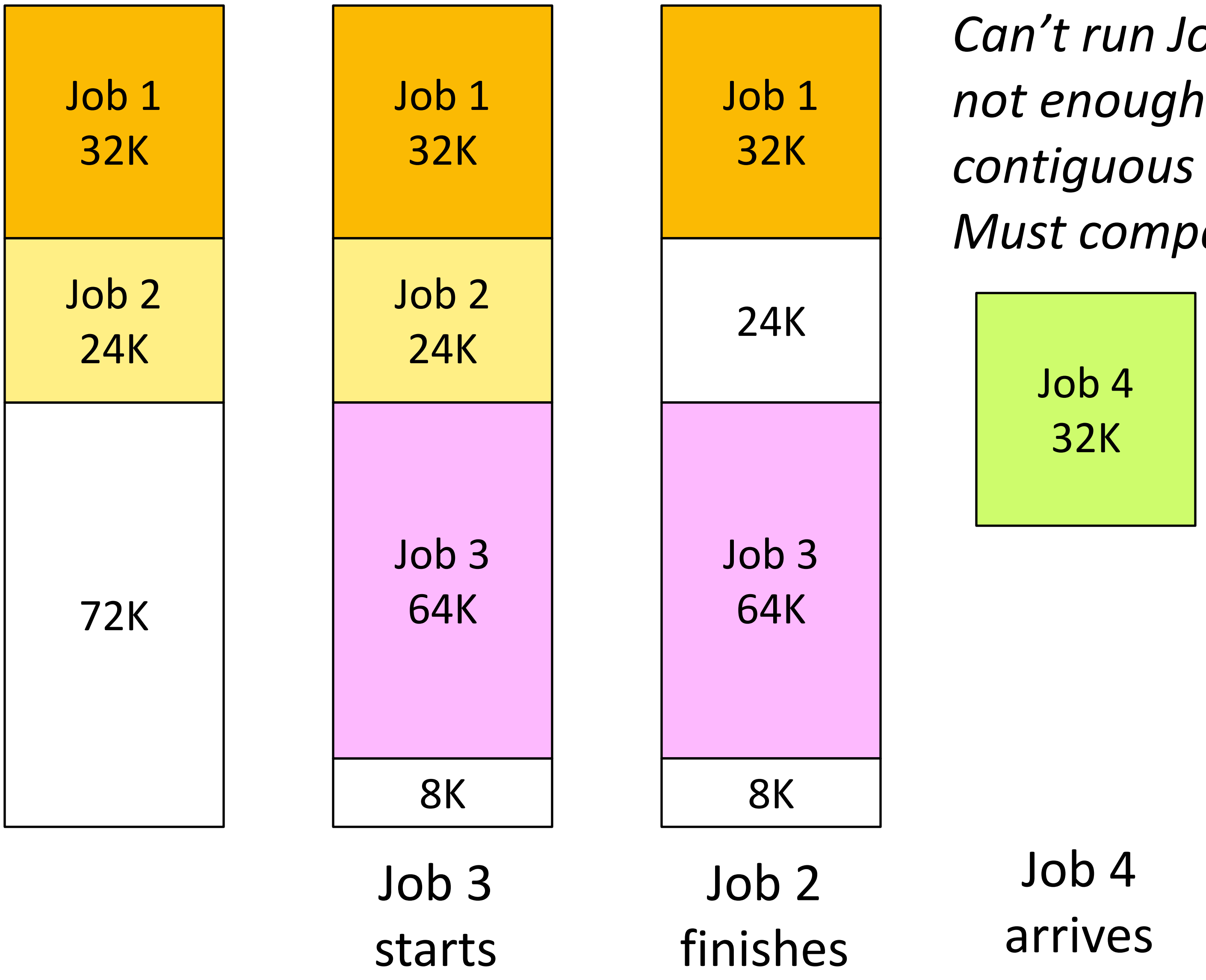
Base and Bound Machine



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

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

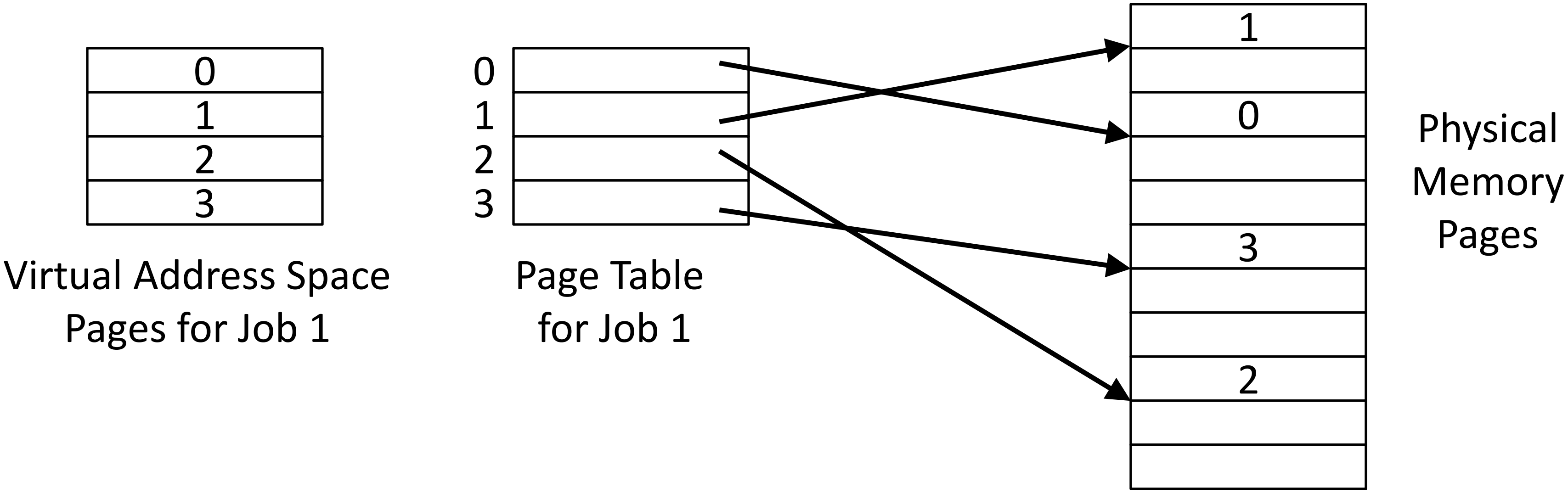


Paged Memory Systems

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

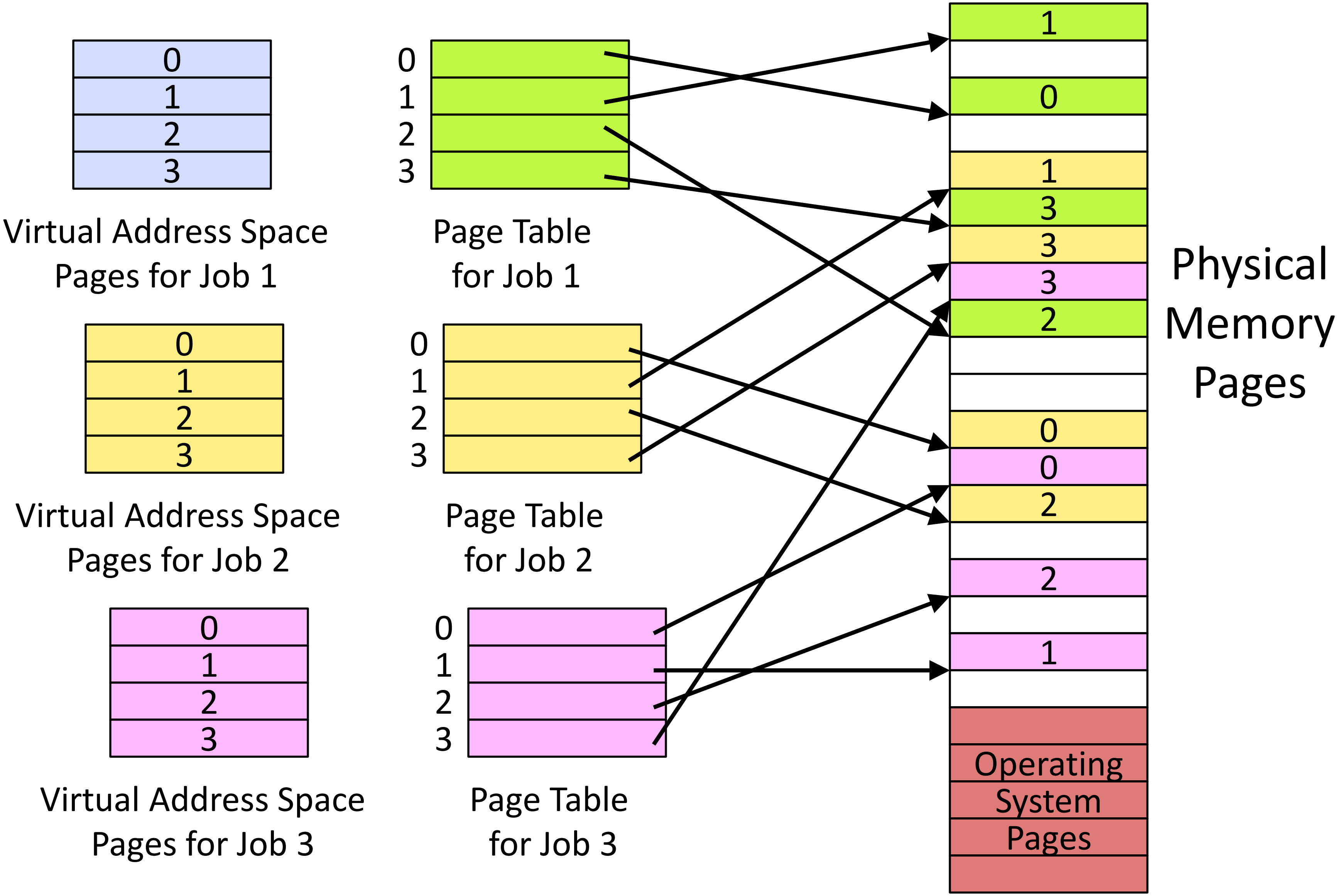
Page Number	Offset

- 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



Virtual Address Space Pages for Job 1

Page Table for Job 1

Virtual Address Space Pages for Job 2

Page Table for Job 2

Virtual Address Space Pages for Job 3

Page Table for Job 3

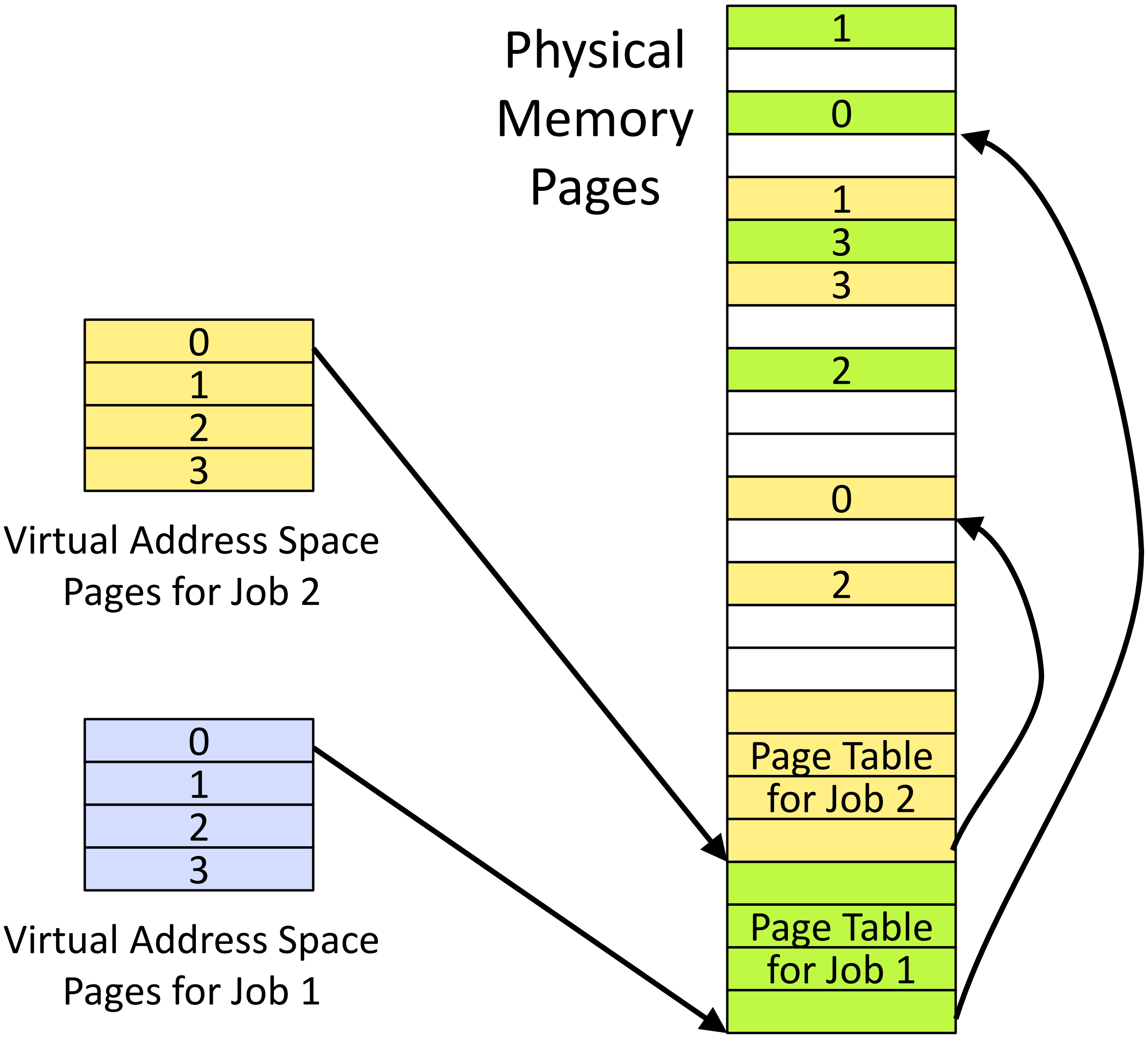
Physical Memory Pages

Operating System Pages

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



Simple linear page tables are too large, so hierarchical page tables are commonly used (see later)

Common for modern OS to place page tables in kernel's virtual memory (page tables can be swapped to secondary storage)

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.

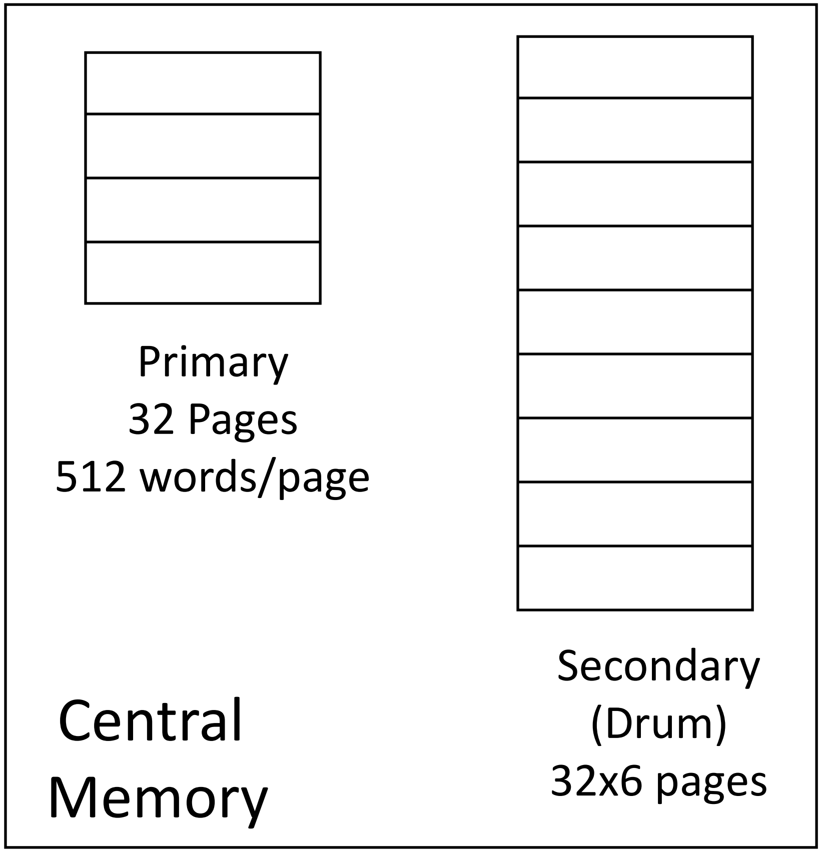
Demand Paging in Atlas (1962)

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

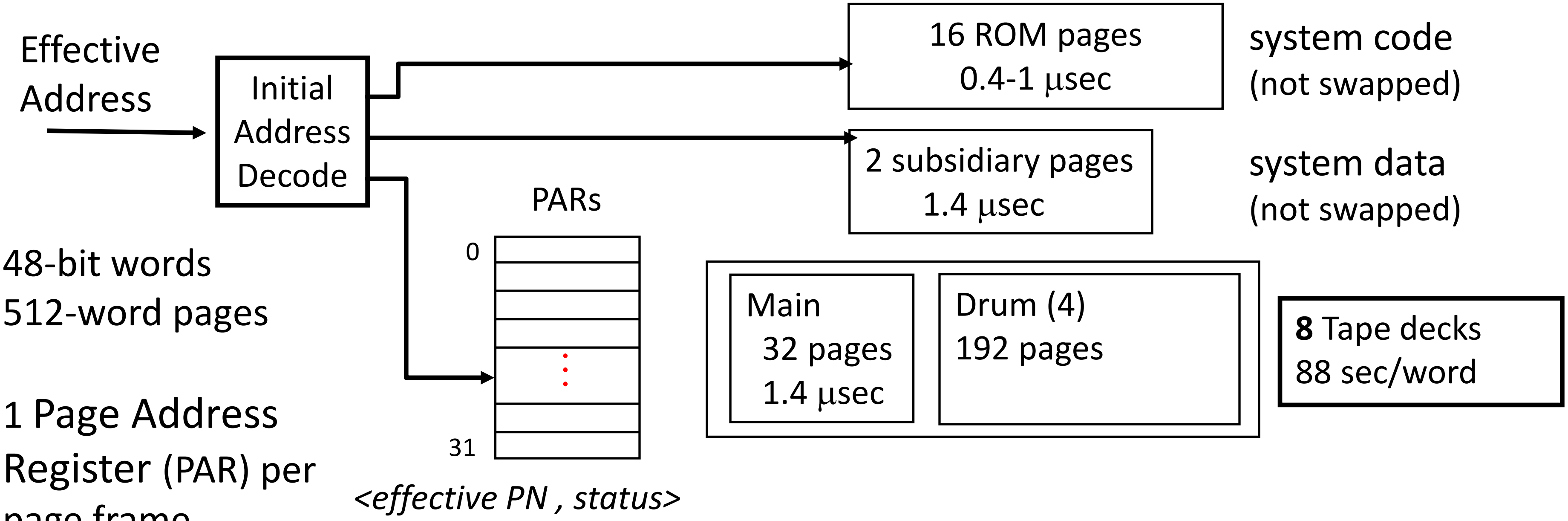
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



Hardware Organization of Atlas



Effective Address

Initial Address Decode

16 ROM pages
0.4-1 μsec

system code
(not swapped)

2 subsidiary pages
1.4 μsec

system data
(not swapped)

PARs

0

31

<effective PN , status>

Main
32 pages
1.4 μsec

Drum (4)
192 pages

8 Tape decks
88 sec/word

48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

Compare the effective page address against all 32 PARs

match ⇒ normal access

no match ⇒ *page fault*

save the state of the partially executed instruction

Atlas Demand-Paging Scheme

On a page fault:

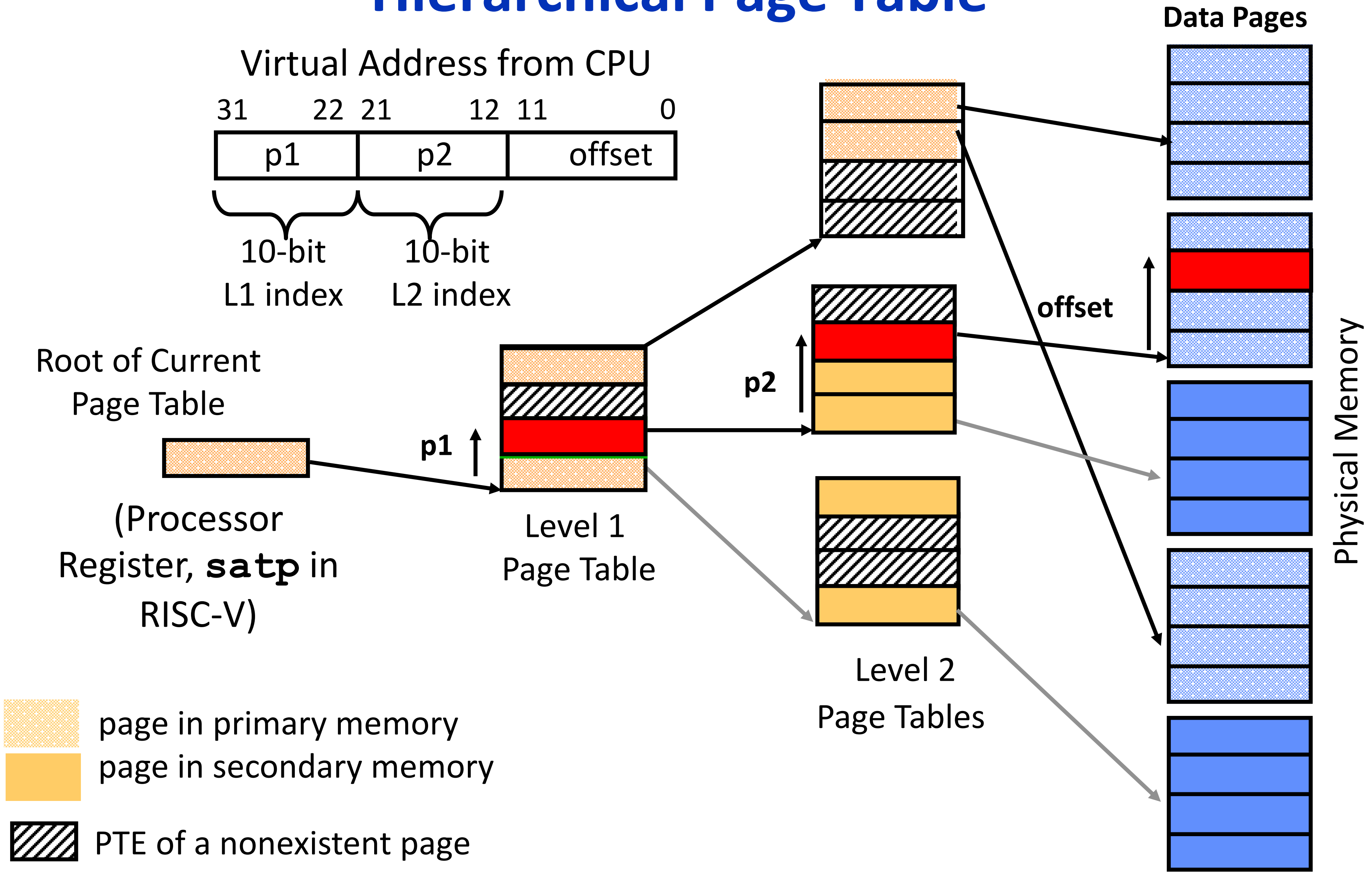
- Input transfer into a free page is initiated
- The Page Address Register (PAR) is updated
- If no free page is left, a page is selected to be replaced (based on usage)
- The replaced page is written on the drum
  - to minimize drum latency effect, the first empty page on the drum was selected
- The 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:
  - 2^20 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!)

What is the "saving grace" ?

Hierarchical Page Table



Data Pages

Virtual Address from CPU

31 22 21 12 11 0

p1 | p2 | offset

10-bit L1 index

10-bit L2 index

Root of Current Page Table

(Processor Register, **`satp`** in RISC-V)

p1

Level 1 Page Table

p2

Level 2 Page Tables

offset

Physical Memory

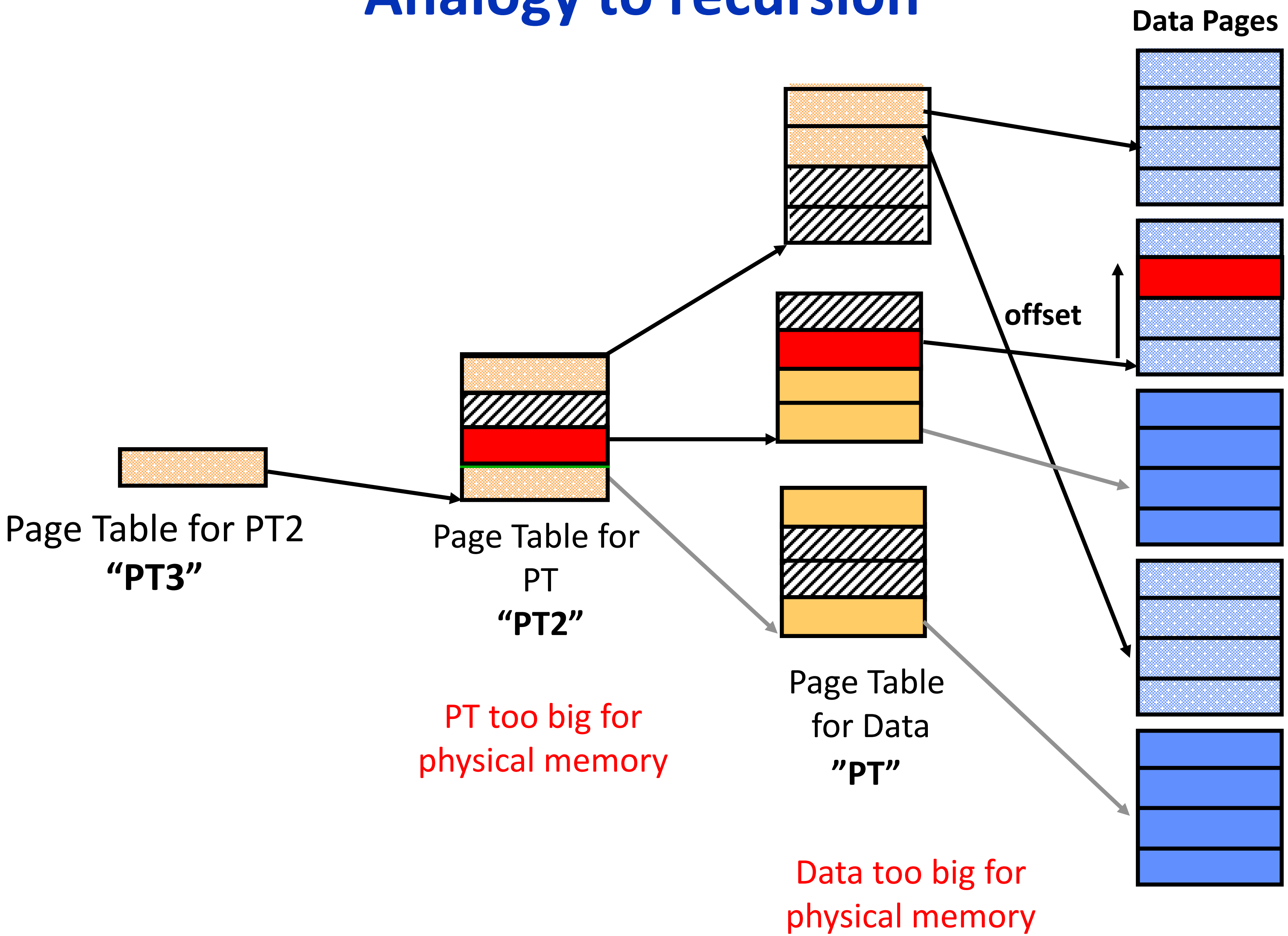
▒ page in primary memory

🟧 page in secondary memory

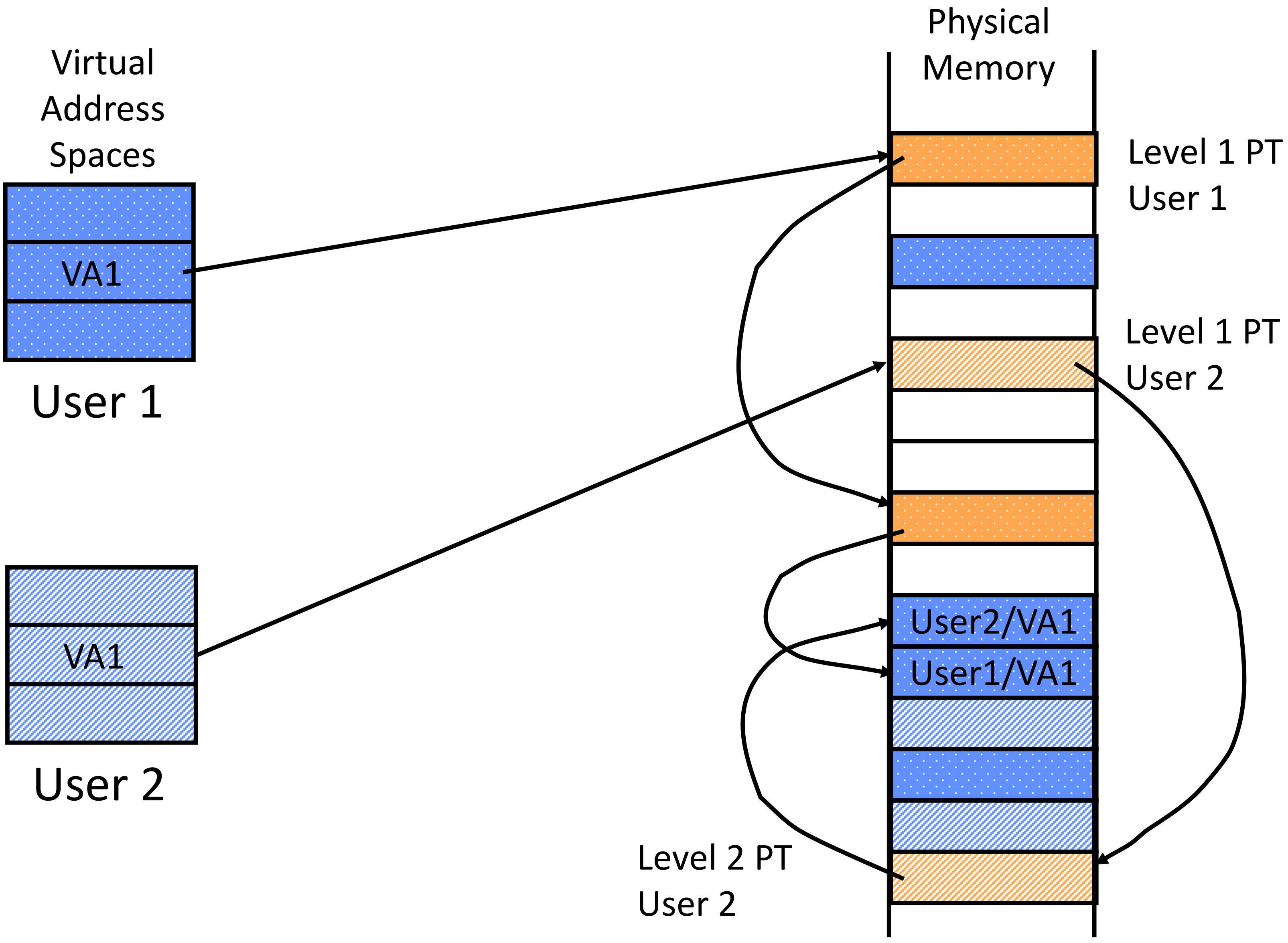
▨ PTE of a nonexistent page

RISC-V Sv32 Virtual Memory Scheme

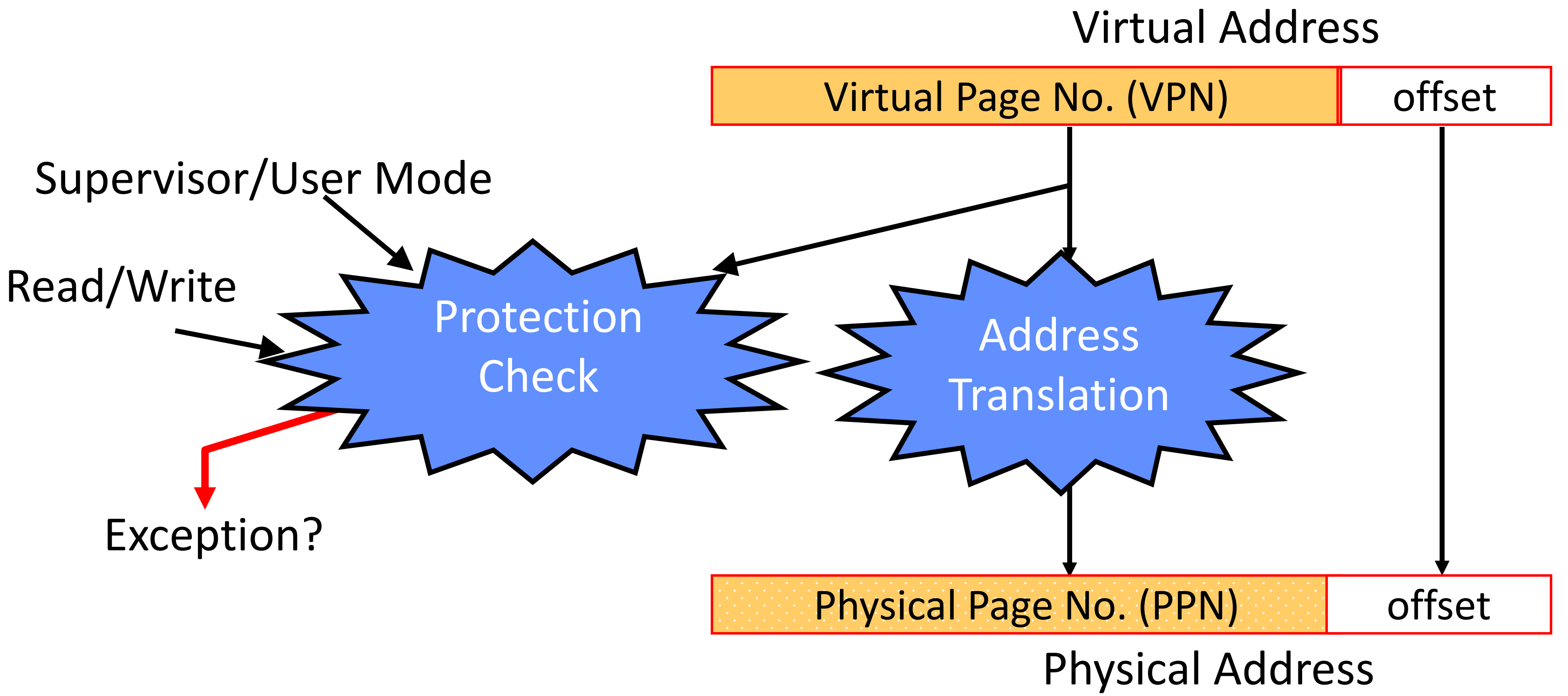
Analogy to recursion



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

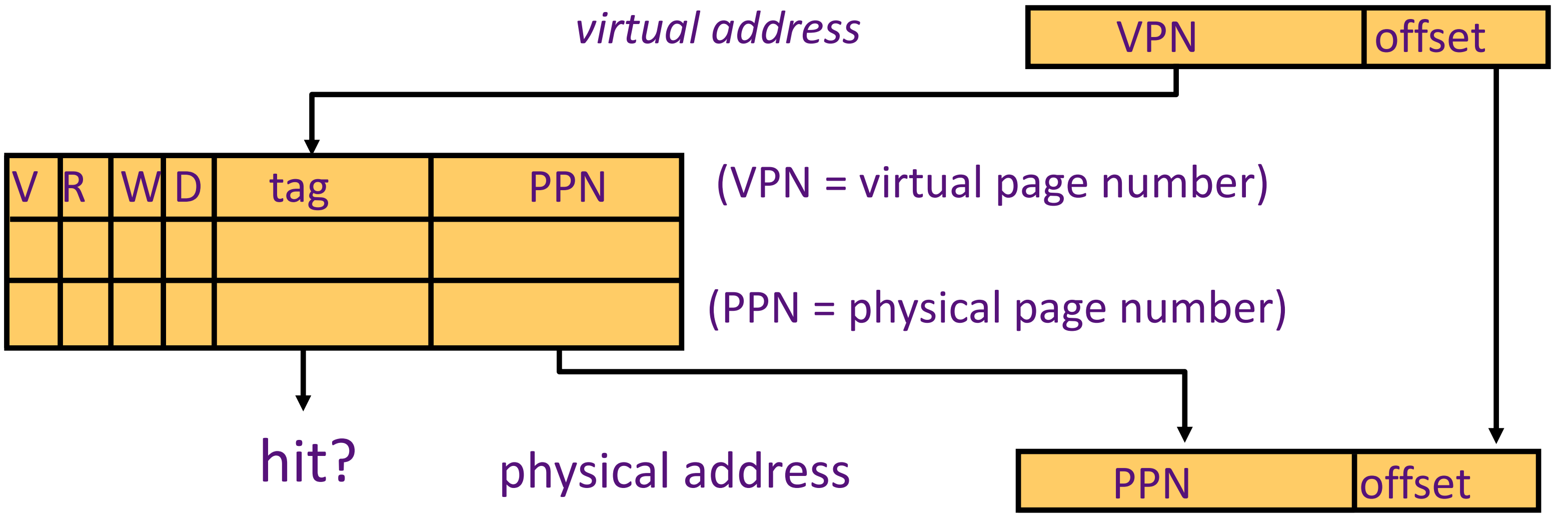
Translation-Lookaside Buffers (TLB)

Address translation is very expensive!

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

Solution: *Cache translations in TLB*

TLB hit	⇒ *Single-Cycle Translation*
TLB miss	⇒ *Page-Table Walk to refill*



virtual address

VPN offset

V R W D tag PPN

(VPN = virtual page number)

(PPN = physical page number)

hit?

physical address

PPN offset

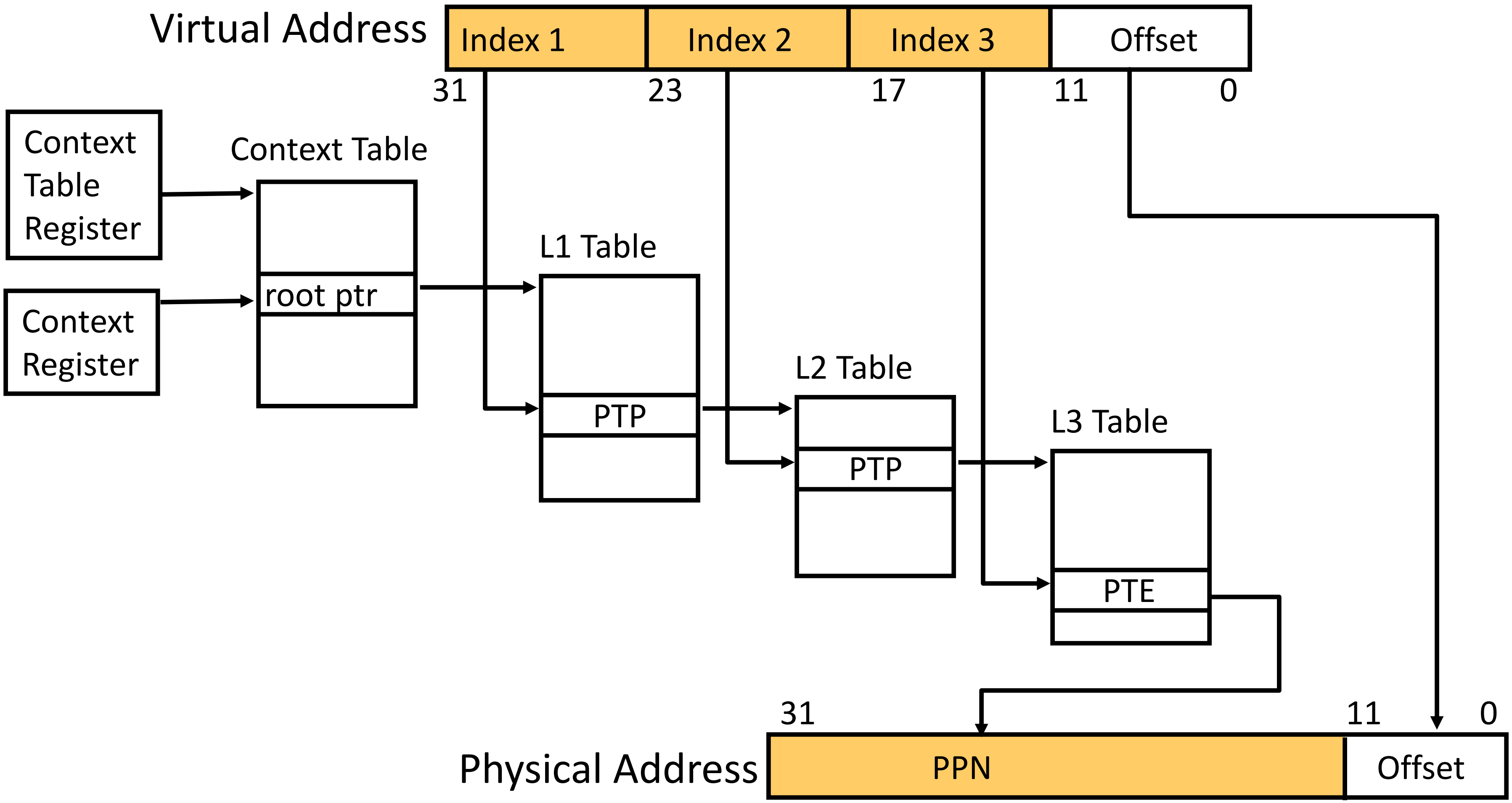
TLB Designs

- 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 entries * 4 KB = 256 KB (if contiguous)*_______________?

Handling a TLB Miss

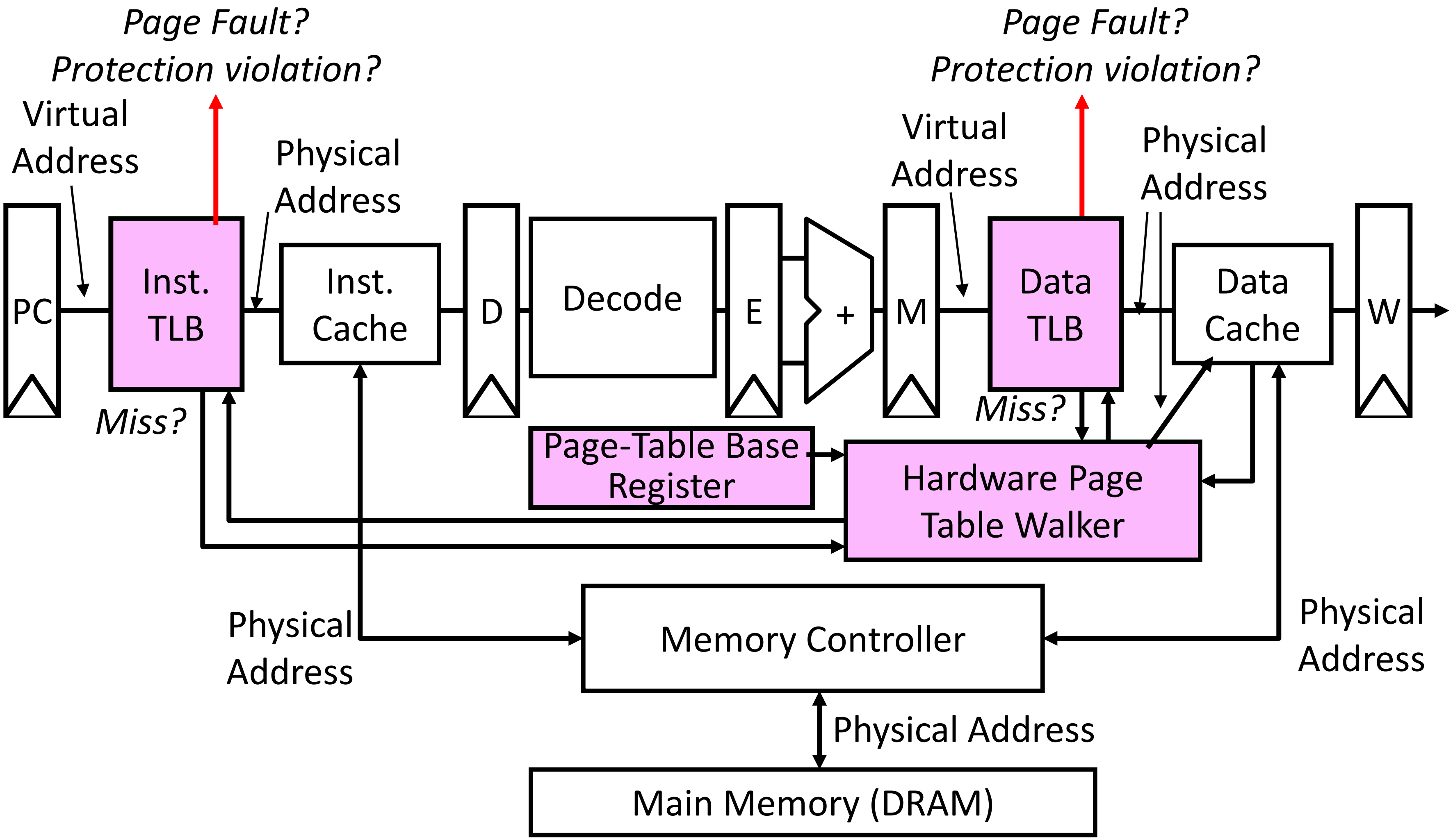
- Software (*MIPS, Alpha*)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged "untranslated" addressing mode used for walk.
  - Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump 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)

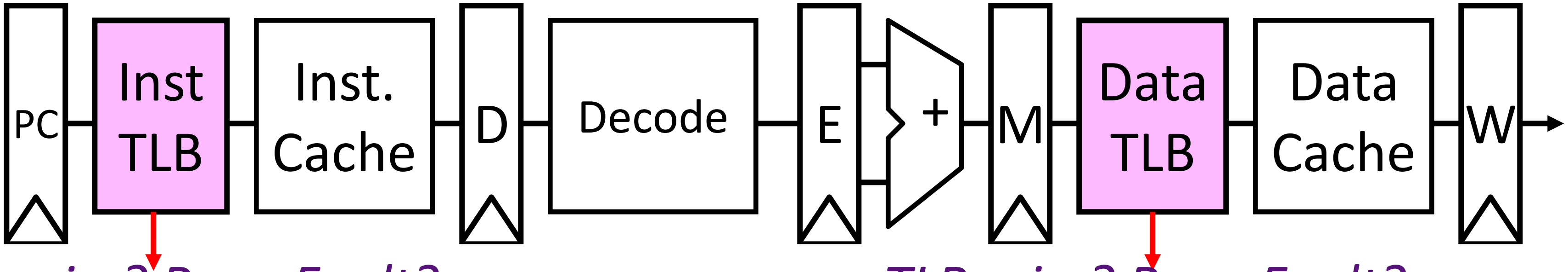


- Assumes page tables held in untranslated physical memory

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 shootdown”
  - Interrupt other processors to invalidate stale TLB entries
  - Some mainframes had hardware TLB coherence

Handling VM-related exceptions



TLB miss? Page Fault? Protection violation?

TLB miss? Page Fault? Protection violation?

- 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

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)
  - Krste Asanovic (UCB)
  - Sophia Shao (UCB)