CSE 530

Lecture 18

Virtual Memory

Fall 2019

Prof. Aasheesh Kolli

Course website: https://sites.psu.edu/akolli

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Kolli, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Pennsylvania State University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Two parts to Modern VM

- *VM provides each process with the illusion of a* *<u>large, private, uniform</u> memory*
- Part A: Protection
  - ❒ each process sees a large, contiguous memory segment without holes
  - ❒ each process's memory space is private, i.e. protected from access by other processes
- Part B: Demand Paging
  - ❒ capacity of secondary memory (swap space on disk)
  - ❒ at the speed of primary memory (DRAM)
- Based on a common HW mechanism: address translation
  - ❒ user process operates on "virtual" or "effective" addresses
  - ❒ HW translates from virtual to physical on each reference
    - ❍ controls which physical locations can be named by a process
    - ❍ allows dynamic relocation of physical backing store (DRAM vs. HD)
  - ❒ VM HW and memory management policies controlled by the OS

Evolution of Protection Mechanisms

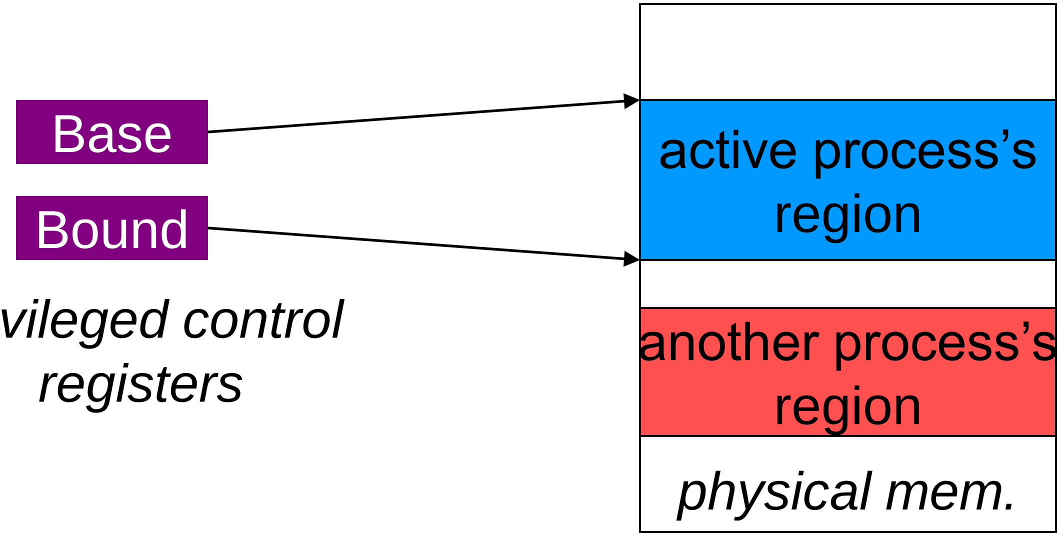
- Earliest machines had no concept of protection and address translation
  - ❐ no need---single process, single user
  - ❐ automatically “private and uniform” *(but not very large)*
  - ❐ programs operated on physical addresses directly

    no multitasking protection, no dynamic relocation

    (at least not very easily)

Base and bound registers

- In a multi-tasking system
- Each process is given a non-overlapping, contiguous physical memory region, *everything belonging to a process must fit in that region*
- When a process is swapped in, OS sets base to the start of the process's memory region and bound to the end of the region
- HW translation and protection check *(on each memory reference)*
- PA = EA + base
- provided (PA < bound), else violations
- ⇒ *Each process sees a private and uniform address space (0 .. max)*



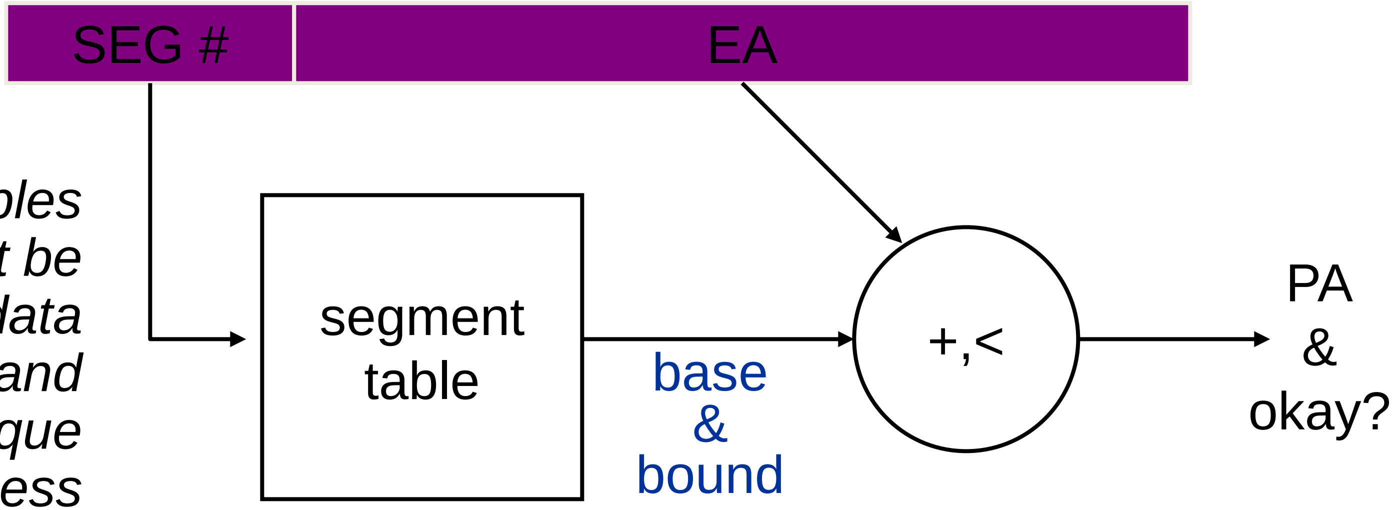
privileged control registers

Bound can also be formulated as a range

Segmented Address Space

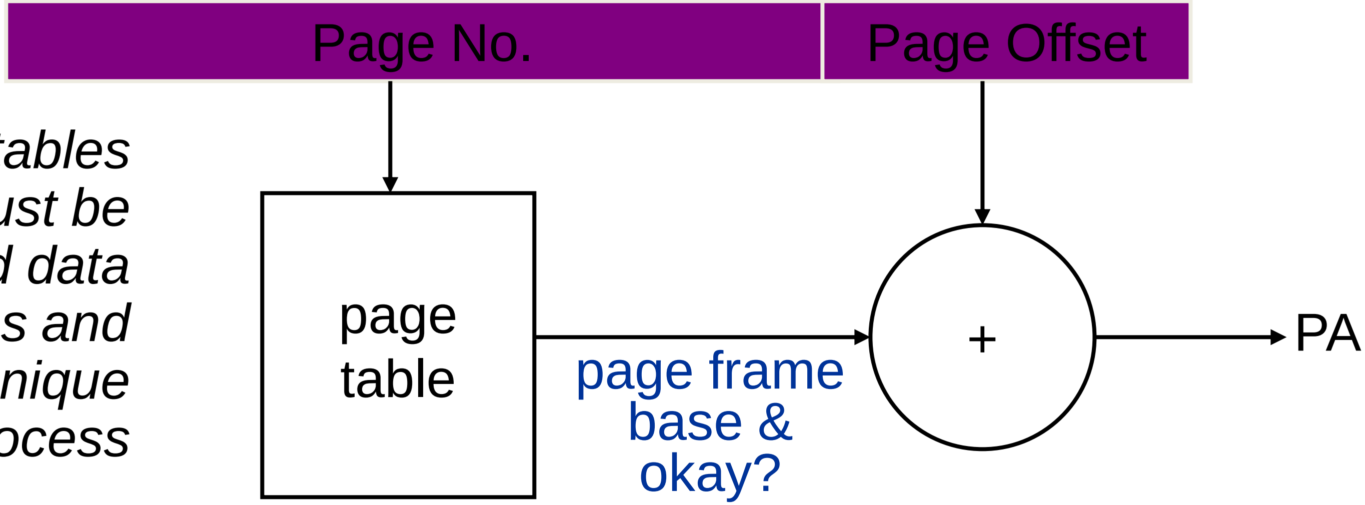
- segment == a base and bound pair
- segmented addressing gives each process multiple segments
  - ❒ initially, separate code and data segments
    - *2 sets of base-and-bound reg's for inst and data fetch*
    - *allowed sharing code segments*
  - ❒ became more and more elaborate: *code, data, stack, etc.*
  - ❒ also (ab)used as a way for an ISA with a small EA space to address a larger physical memory space

segment tables must be
1. privileged data structures and
2. private/unique to each process



Paged Address Space

- Segmented addressing creates fragmentation problems,
  - ❒ a system may have plenty of unallocated memory locations
  - ❒ they are useless if they do not form a <u>contiguous</u> region of a sufficient size
- In a Paged Memory System:
- PA space is divided into fixed size segments (e.g. 4kbyte), more commonly known as “page frames”
- EA is interpreted as page number and page offset
-



page tables must be
1. privileged data structures and
2. private/unique to each process

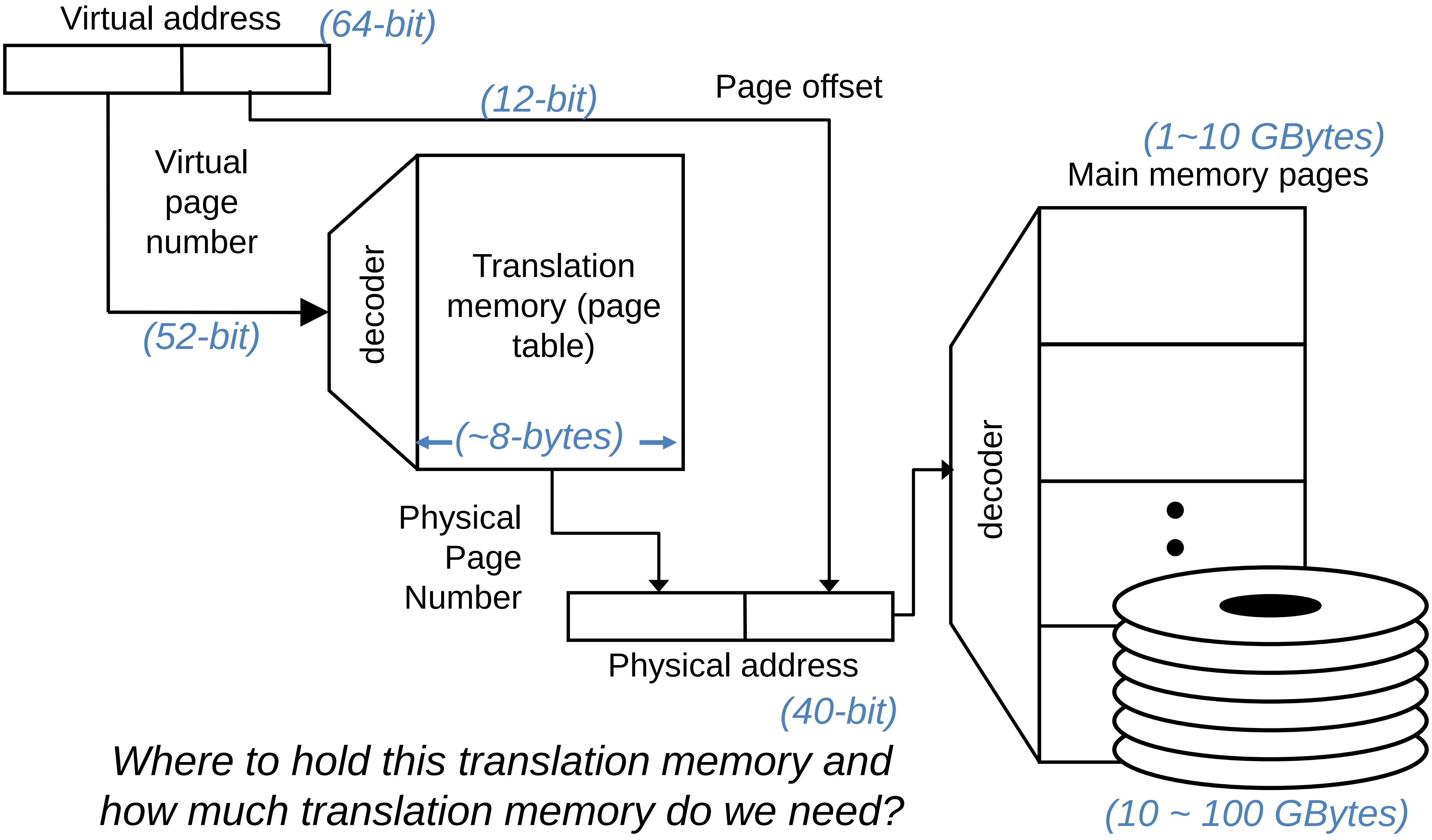
Demand Paging

- Main memory and Disk as automatically managed levels in the memory hierarchies
- *analogous to cache vs. main memory*
- Drastically different size and time scales
- ⇒ very different design decisions

Demand Paging vs. Caching: 2016

	Cache	Demand Paging
• capacity	64KB~MB	??
		1GB~1TB ??
• block size	16~128 Byte	4K to 64K Byte
• hit time	1~3 cyc	50-150 cyc
• miss penalty	10~300 cycles	1M to 10M cycles
• miss rate	0.1~10%	0.00001~0.001%
• hit handling	hw	hw
• miss handling	hw	sw

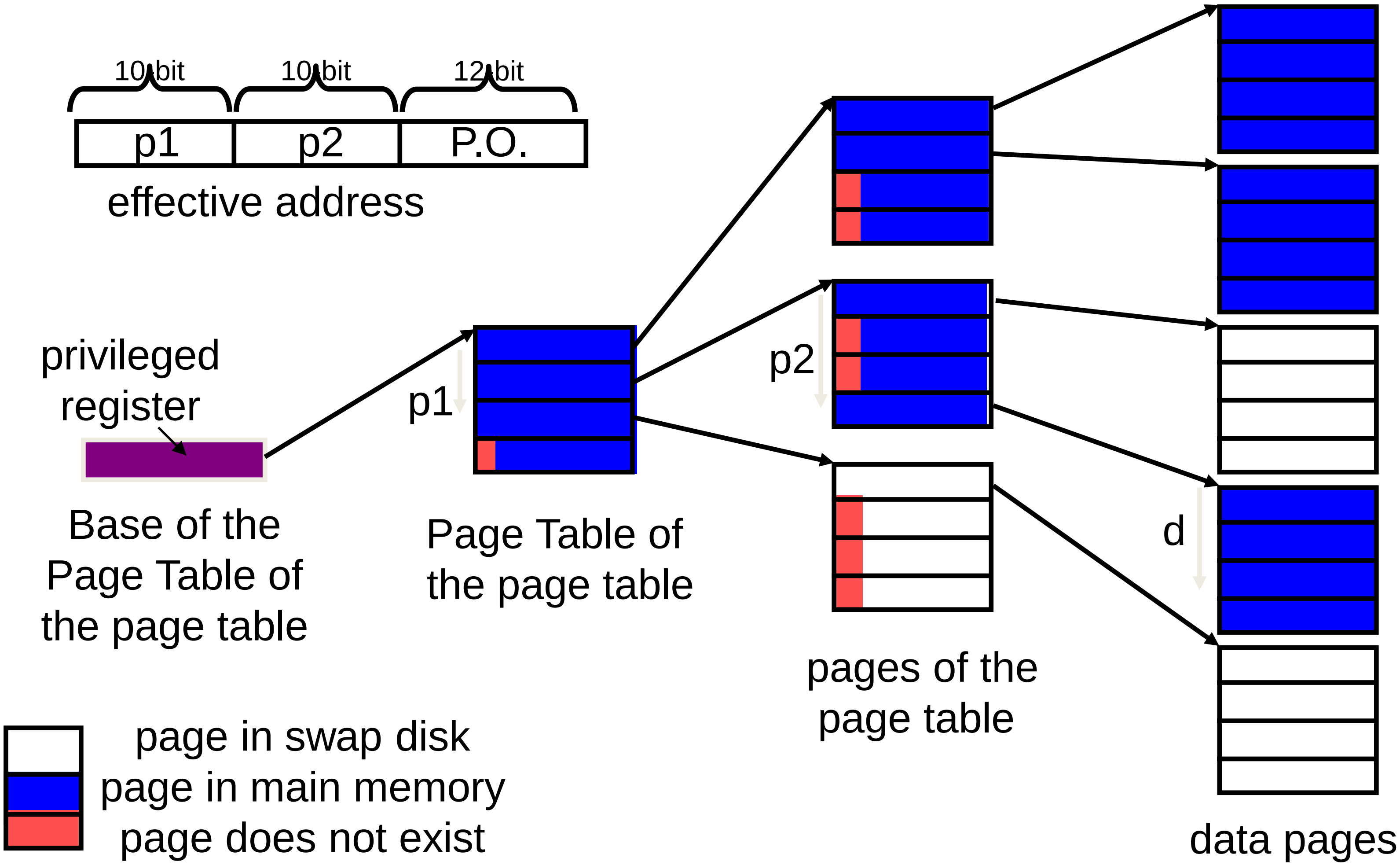
Page-Based Virtual Memory



Where to hold this translation memory and how much translation memory do we need?

Page table organization

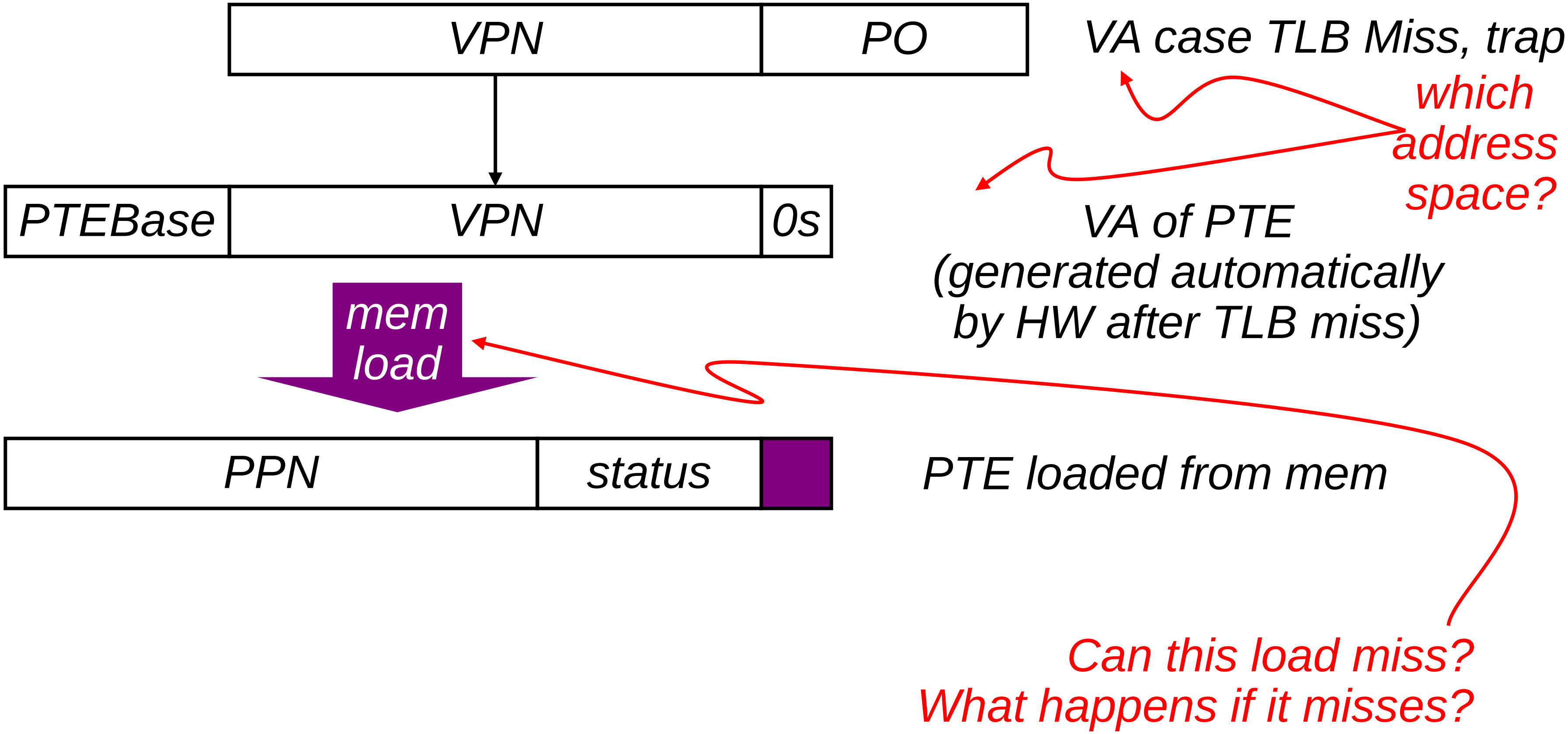
Hierarchical Page Table



Storage of overhead of translation should be proportional to the size of physical memory and not the virtual address space

Bottom-Up Hierarchical Table (MIPS)

- Page table organization is not part of the ISA
- Reference design optimized for software TLB miss handling



VA case TLB Miss, trap

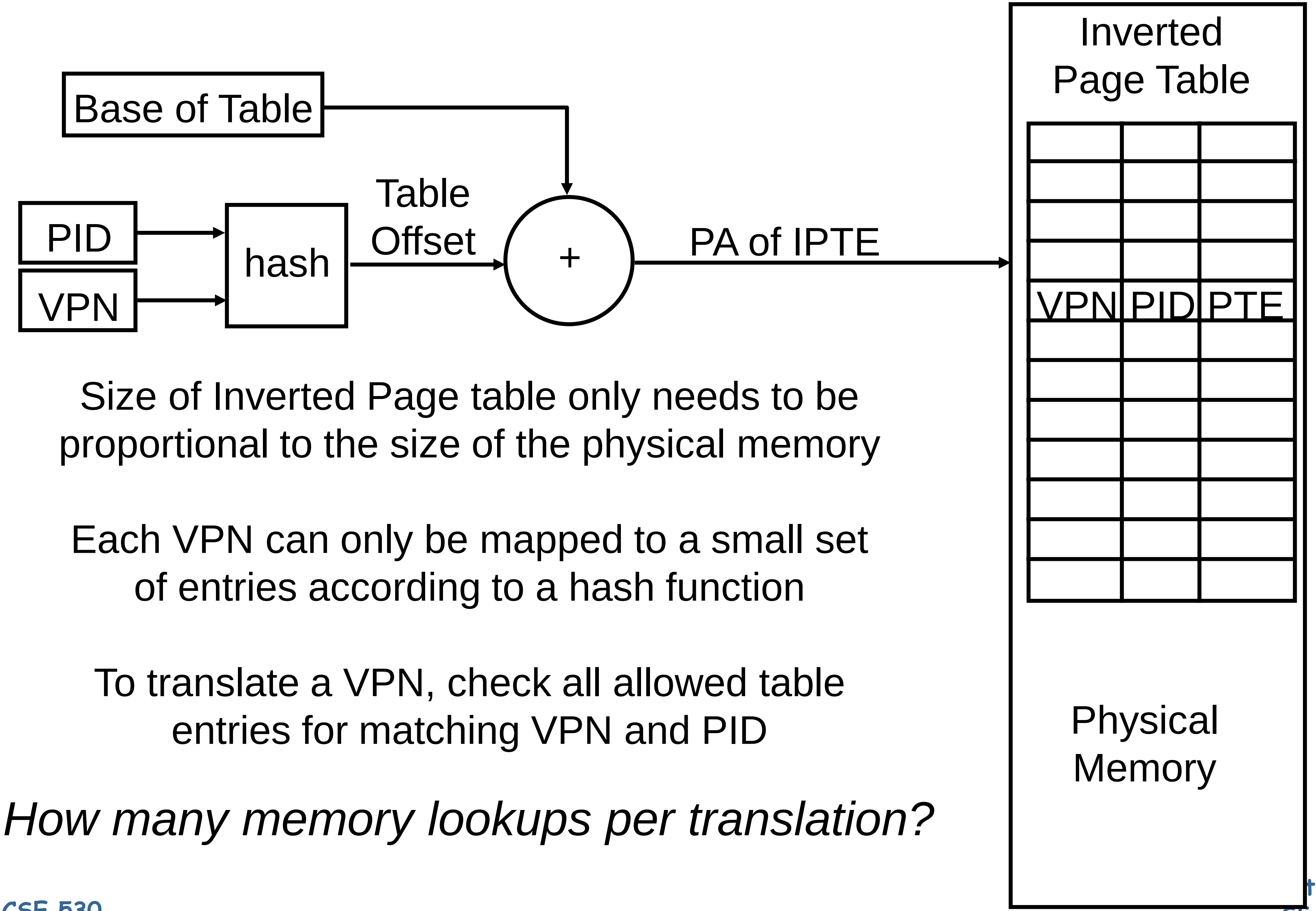
which address space?

VA of PTE (generated automatically by HW after TLB miss)

PTE loaded from mem

Can this load miss?
What happens if it misses?

Inverted or Hashed Page Tables



Size of Inverted Page table only needs to be proportional to the size of the physical memory

Each VPN can only be mapped to a small set of entries according to a hash function

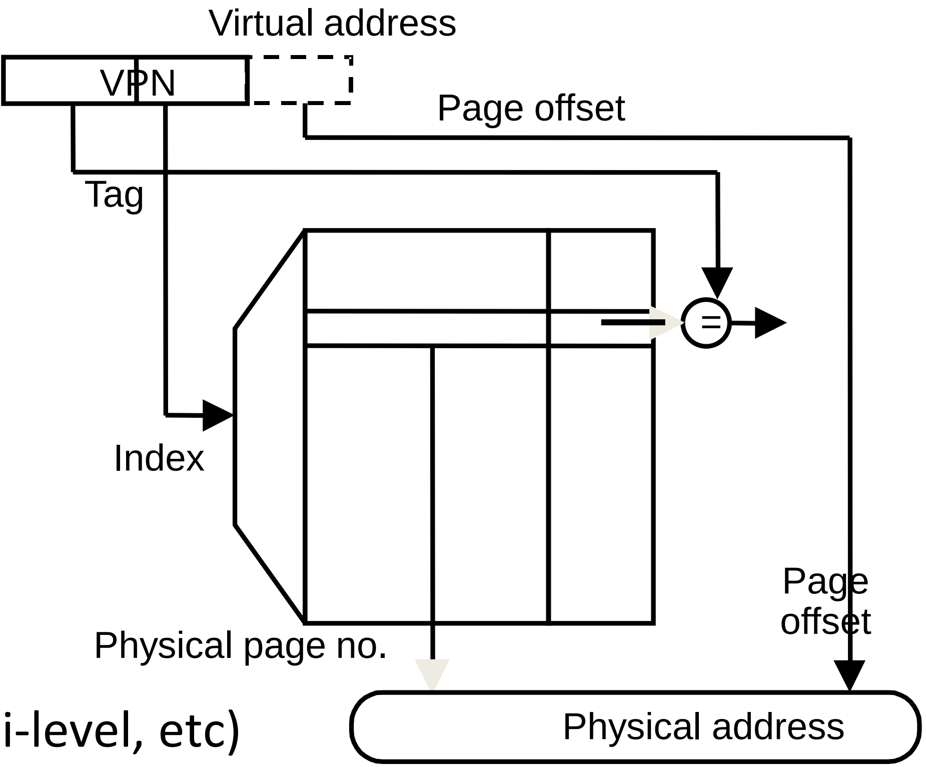
To translate a VPN, check all allowed table entries for matching VPN and PID

How many memory lookups per translation?

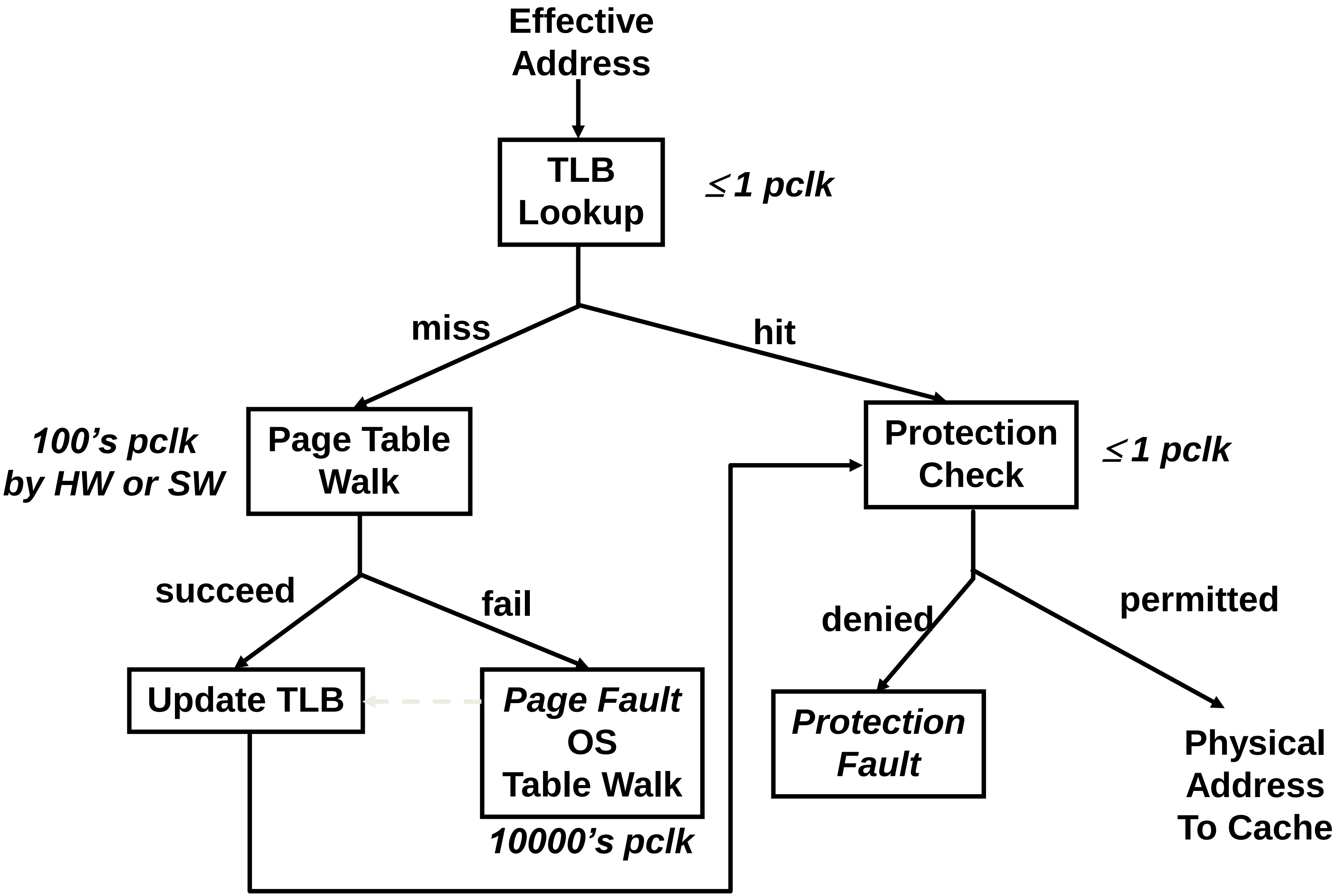
Virtual-to-Physical Translation

Translation Look-aside Buffer (TLB)

- Essentially a cache of
  - ❐ recent address translations
  - ❐ *avoids going to the page table on every reference*
- indexed by lower bits of
- VPN (virtual page #)
- tag = unused bits of VPN + process ID
- data = a page-table entry
  i.e. PPN (physical page #) and
  access permission
- status = valid, dirty
- the usual cache design choices
  (placement, replacement policy multi-level, etc)
  apply here too.

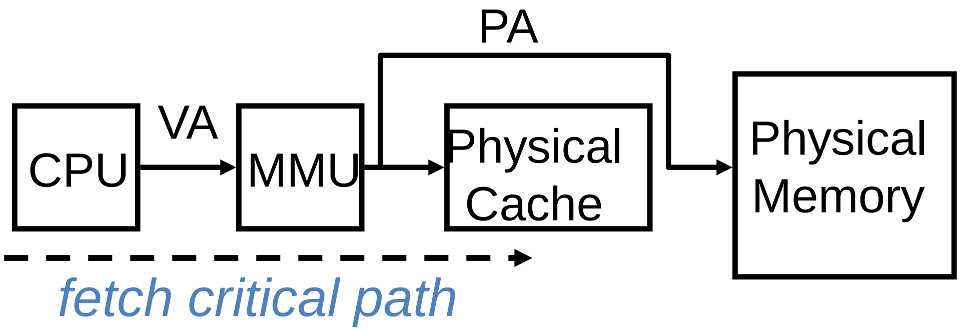


Virtual to Physical Address Translation



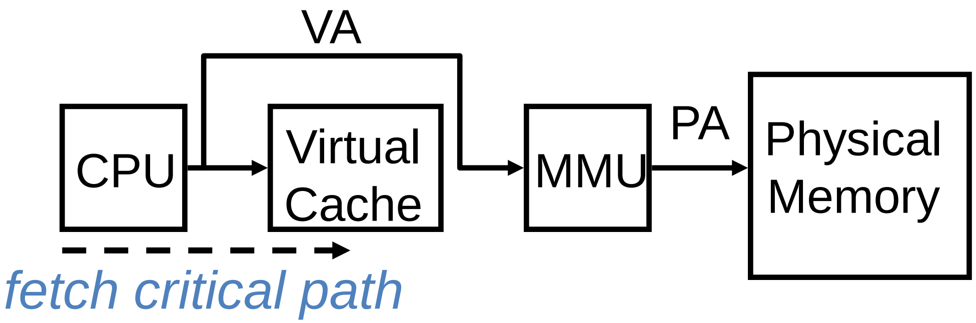
Cache Placement and Address Translation

Physical Cache (Most Systems)



longer hit time

Virtual Cache (SPARC2's)

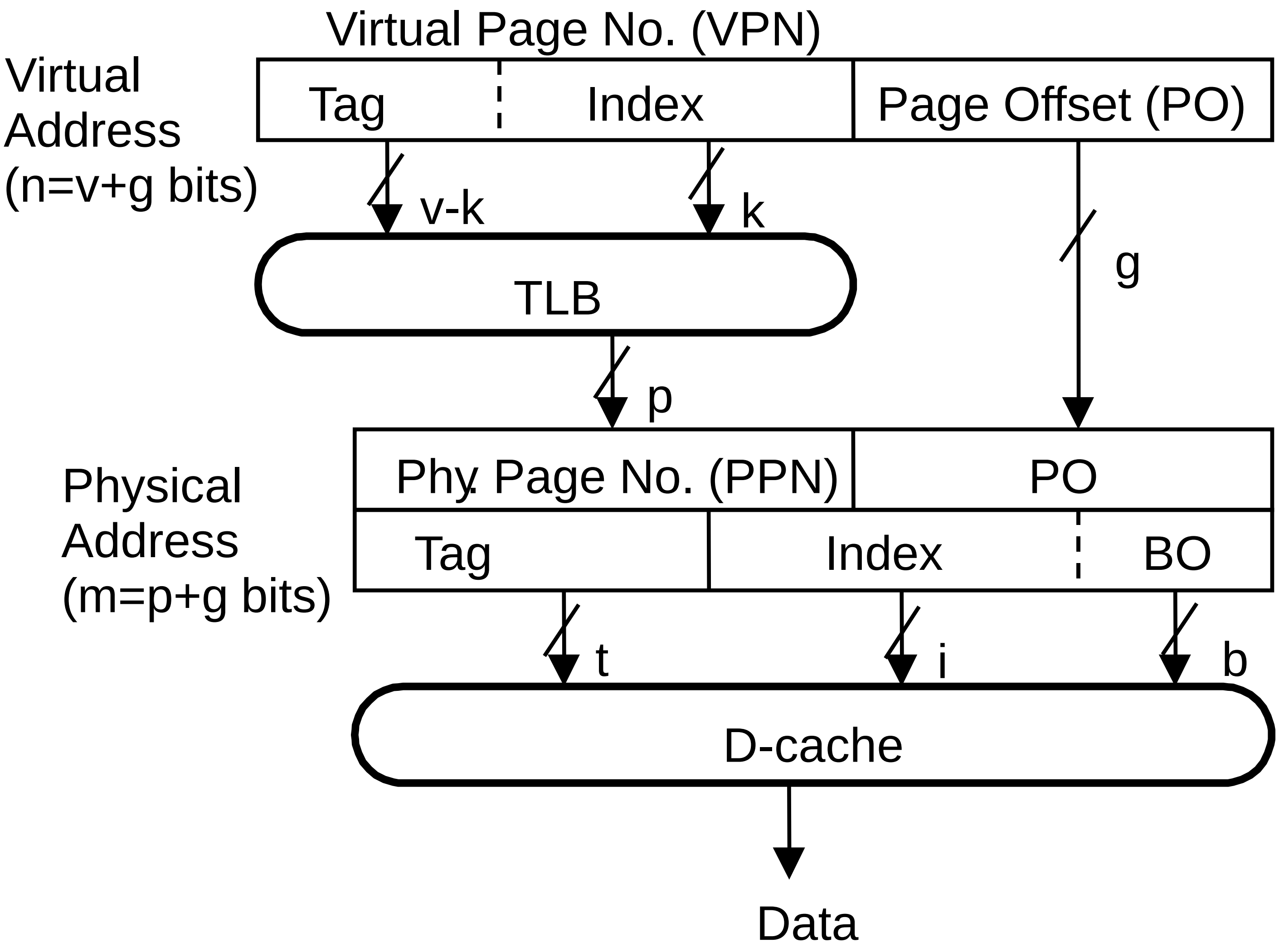


aliasing problem

cold start after context switch

Virtual caches are not popular anymore because MMU and CPU can be integrated on one chip

Physically Indexed Cache



Virtual Page No. (VPN)

Virtual
Address
(n=v+g bits)

Tag | Index | Page Offset (PO)

v-k

k

g

TLB

p

Physical
Address
(m=p+g bits)

Phy. Page No. (PPN) | PO

Tag | Index | BO

t

i

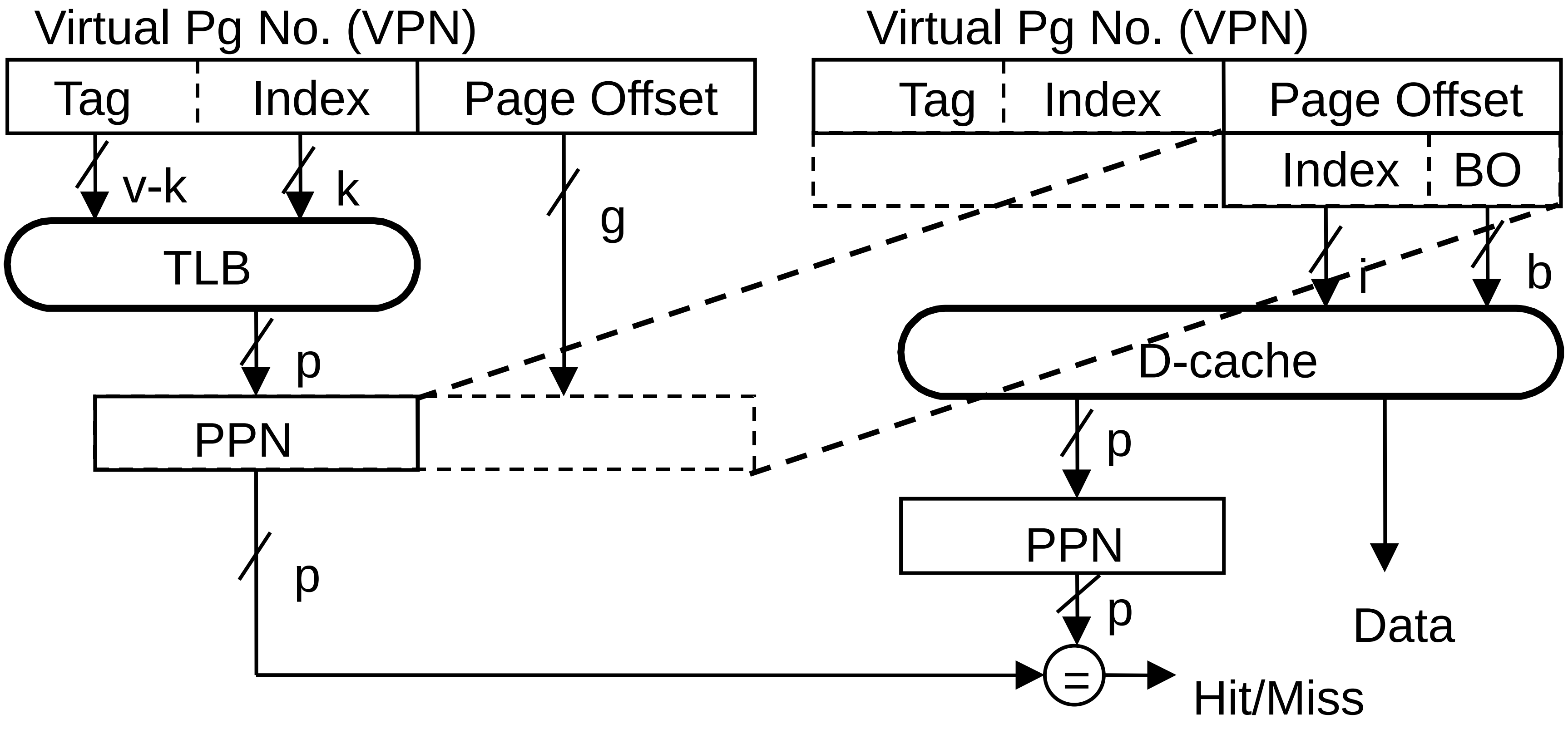
b

D-cache

Data

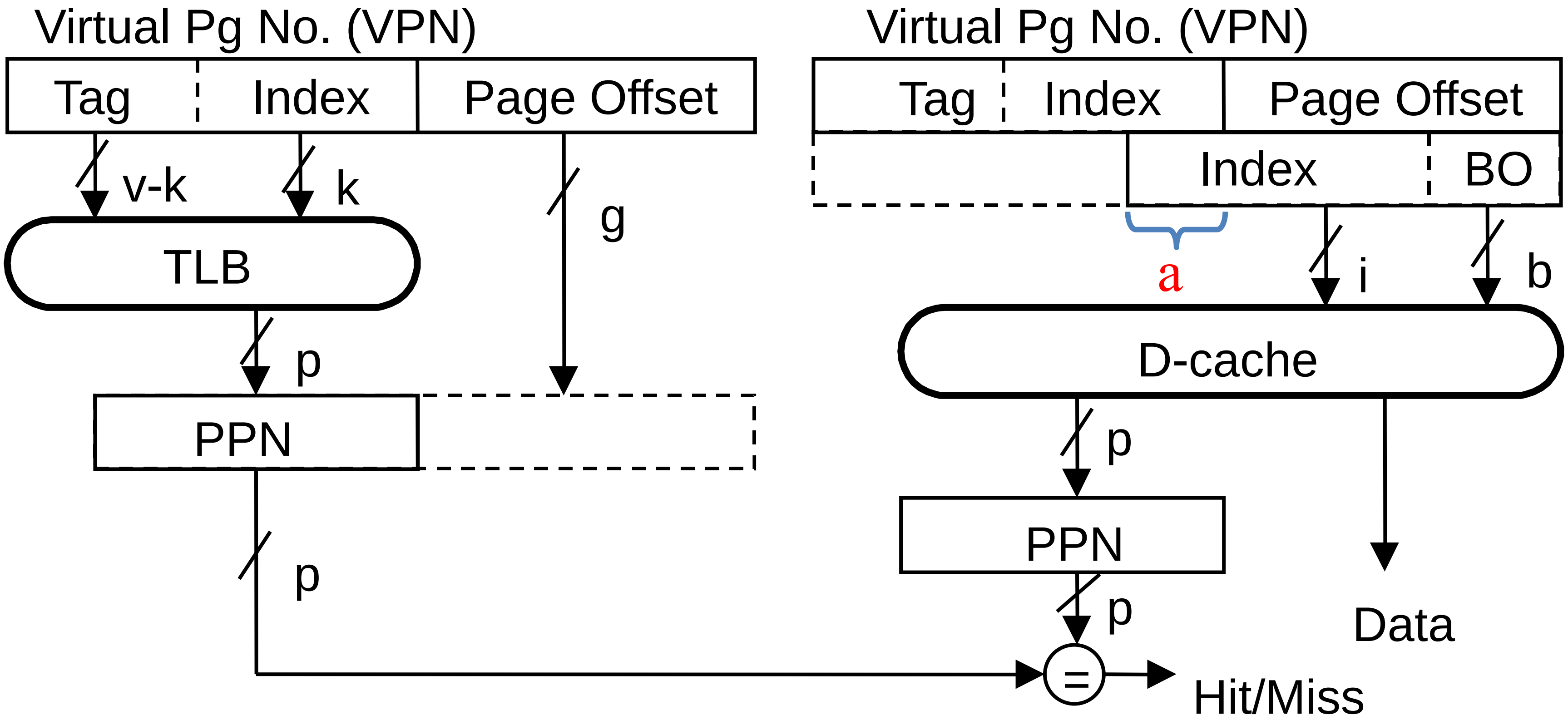
Virtually Indexed Cache

Parallel Access to TLB and Cache arrays



How large can a virtually indexed cache get?

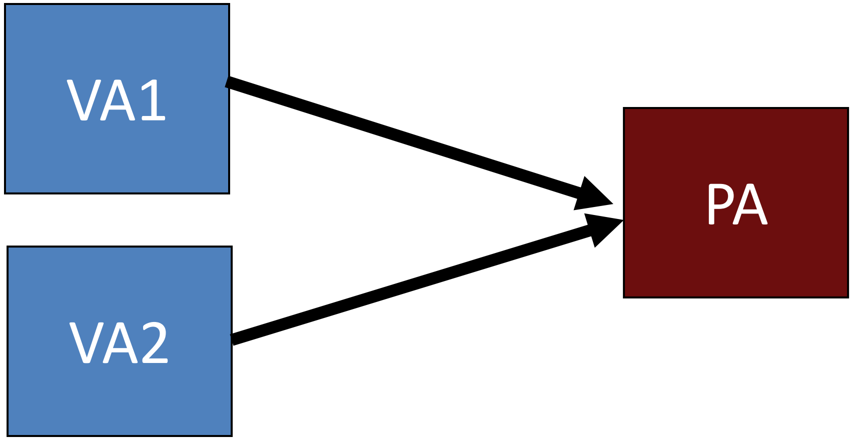
Large Virtually Indexed Cache



If two VPNs differs in *a*, but both map to the same PPN then there is an aliasing problem

Virtual Address Synonyms

- To Virtual pages that map to the same physical page
  - ❒ within the same virtual address space
  - ❒ across address spaces



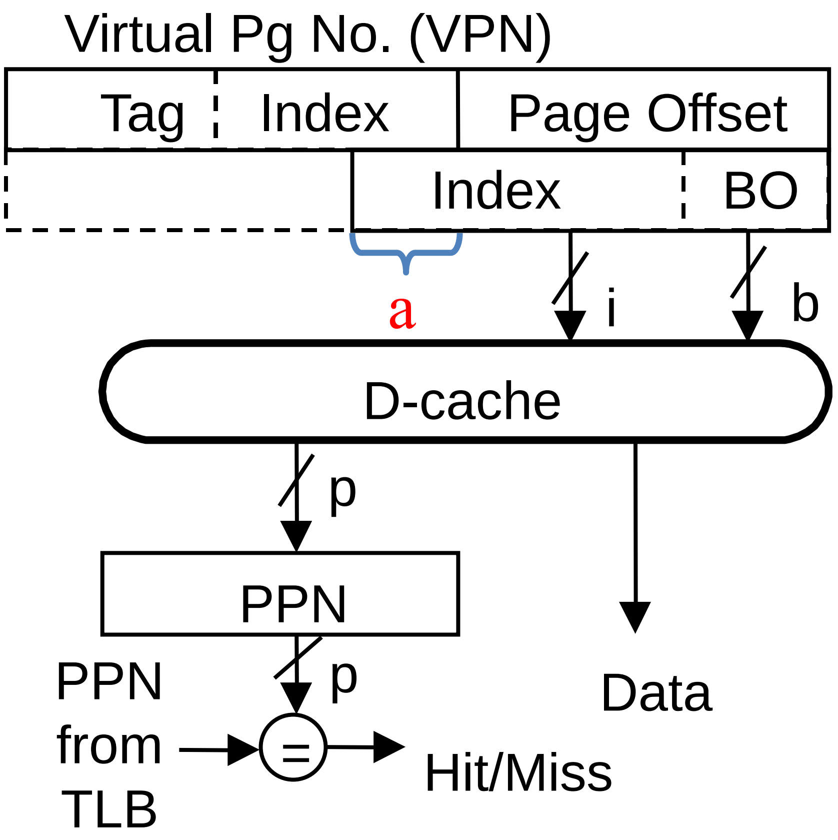
Using VA bits as IDX, PA data may reside in different sets in cache!!

Synonym (or Aliasing)

- When VPN bits are used in indexing, two virtual addresses that map to the same physical address can end up sitting in two cache lines

- In other words, two copies of the same physical memory location may exist in the cache

- ⇒ *modification to one copy won’t be visible in the other*



If the two VPNs do not differ in a then there is no aliasing problem

Synonym Solutions

- Limit cache size to page size times associativity
  - ❒ get index from page offset
- Search all sets in parallel
  - ❒ 64K 4-way cache, 4K pages, search 4 sets (16 entries)
  - ❒ Slow!
- Restrict page placement in OS
  - ❒ make sure index(VA) = index(PA)
- Eliminate by OS convention
  - ❒ single virtual space
  - ❒ restrictive sharing model

MIPS R10K Synonym Solution

- 32KB 2-Way Virtually-Indexed L1
  - ❒ needs 10 bits of index and 4 bits of block offset
  - ❒ page offset is only 12-bits ⇒ 2 bits of index are VPN[1:0]
- Direct-Mapped Physical L2
  - ❒ L2 is *Inclusive* of L1
  - ❒ VPN[1:0] is appended to the “tag” of L2
- Given two virtual addresses *VA* and *VB* that differs in *a* and both map to the same physical address *PA*
  - ❒ Suppose *VA* is accessed first so blocks are allocated in L1&L2
  - ❒ What happens when *VB* is referenced?

    1 *VB* indexes to a different block in L1and misses

    2 *VB* translates to *PA* and goes to the same block as *VA* in L2

    3. Tag comparison fails (*VA*[1:0]≠*VB*[1:0])

    4. L2 detects that a synonym is cached in L1 ⇒ *VA*’s entry in L1 is ejected before *VB* is allowed to be refilled in L1