Modern Virtual Memory Systems

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

A Problem in Early Sixties

- There were many applications whose data could not fit in the main memory, e.g., payroll
  - *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*

- Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

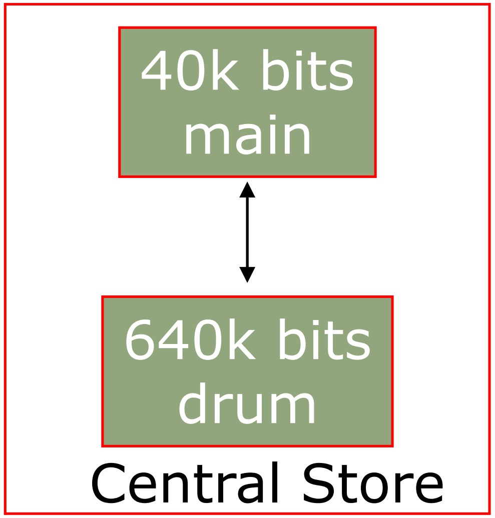
Manual Overlays

- Assume an instruction can address all the storage on the drum

- *Method 1:* programmer keeps track of addresses in the main memory and initiates an I/O transfer when required

- *Method 2:* automatic initiation of I/O transfers by software address translation

  Brooker's interpretive coding, 1960



Ferranti Mercury
1956

Problems?

Method1: Difficult, error prone
Method2: Inefficient

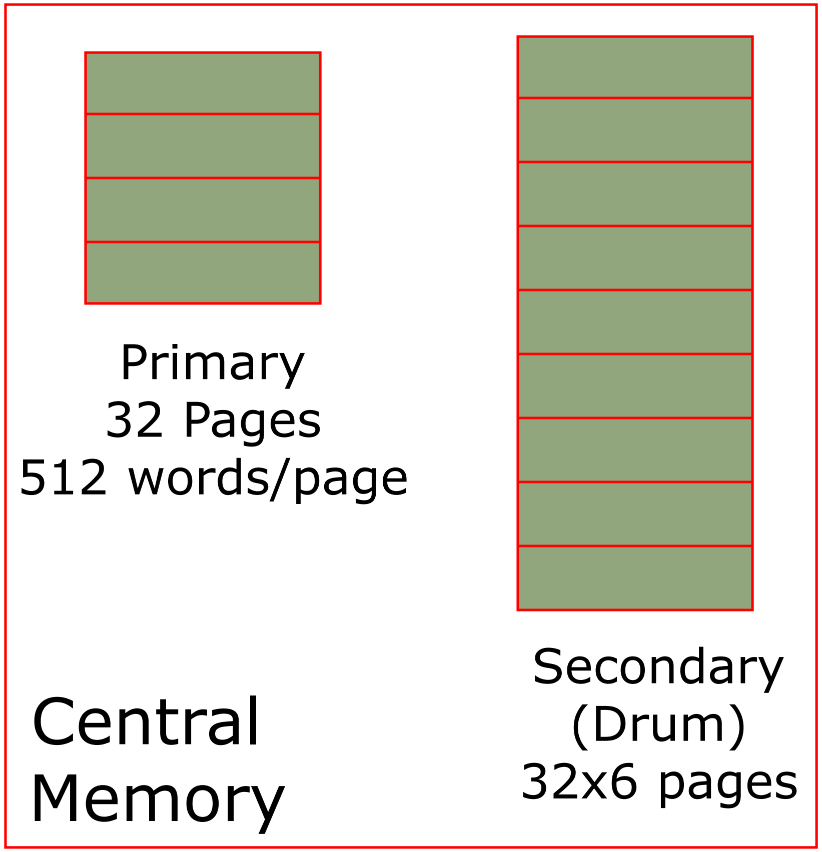
Demand Paging in Atlas (1962)

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

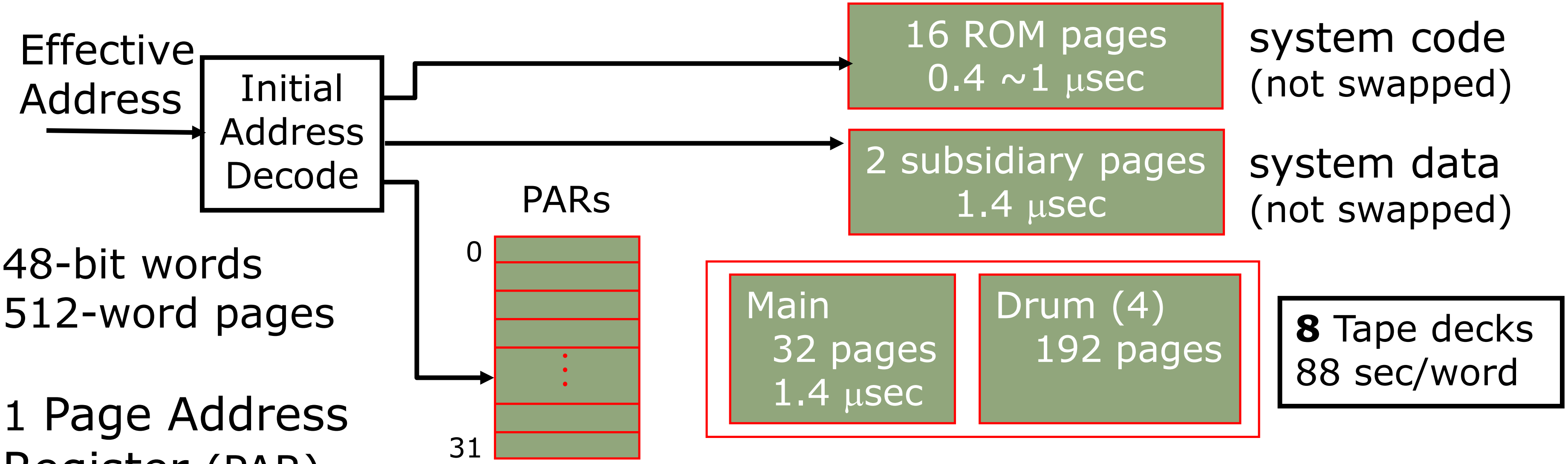
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



Hardware Organization of Atlas



48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

<effective PN , status>

Compare the effective page address against all 32 PARs

match ⇒ normal access

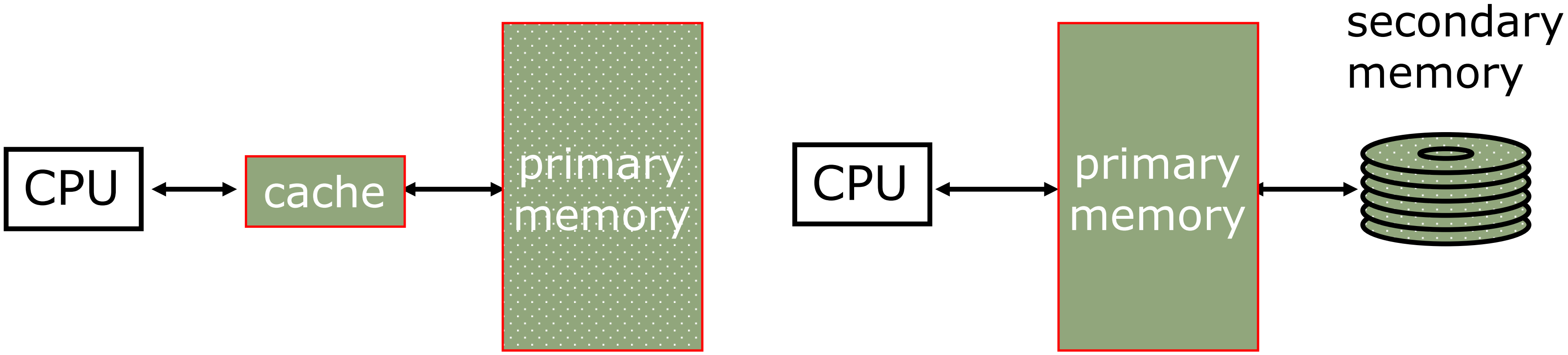
no match ⇒ *page fault*

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 the 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

Caching vs. Demand Paging



Caching
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

Demand paging
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*

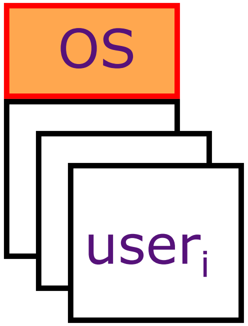
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy

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

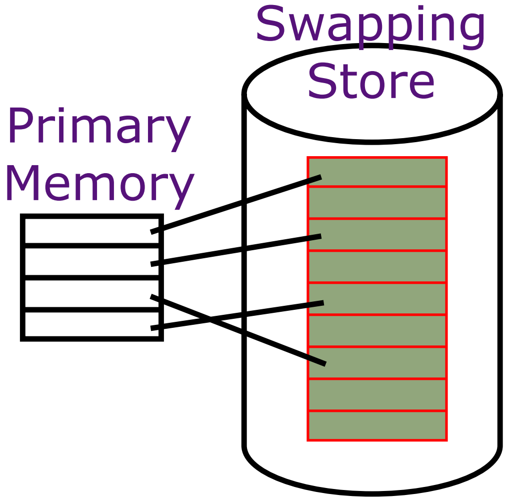
page table ≡ name space



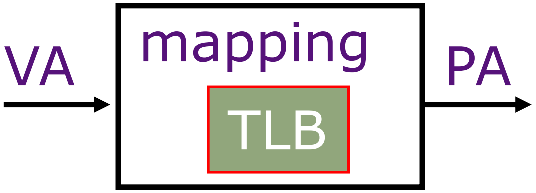
Demand Paging

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

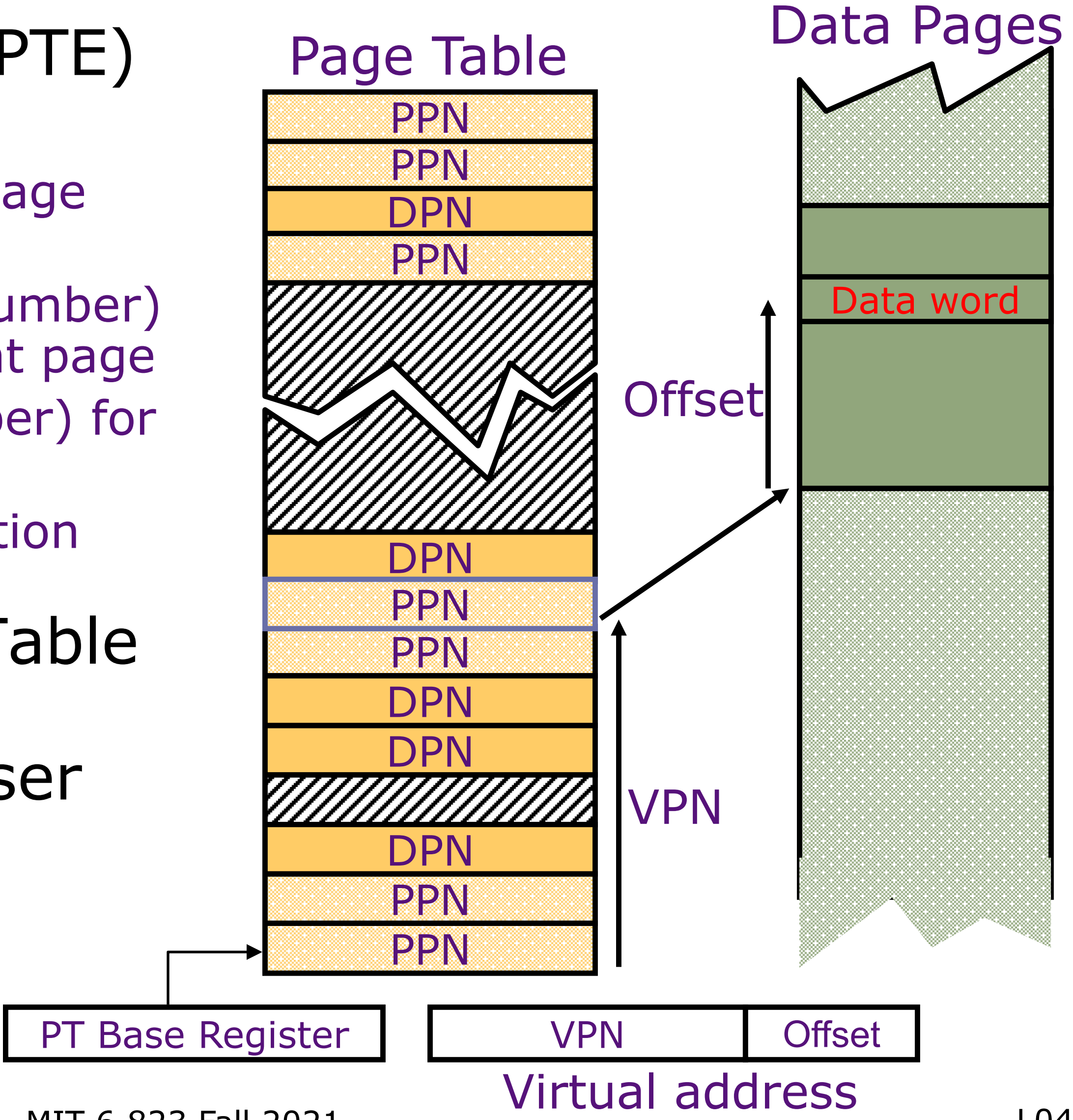


The price is address translation on each memory reference



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

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

Larger pages?

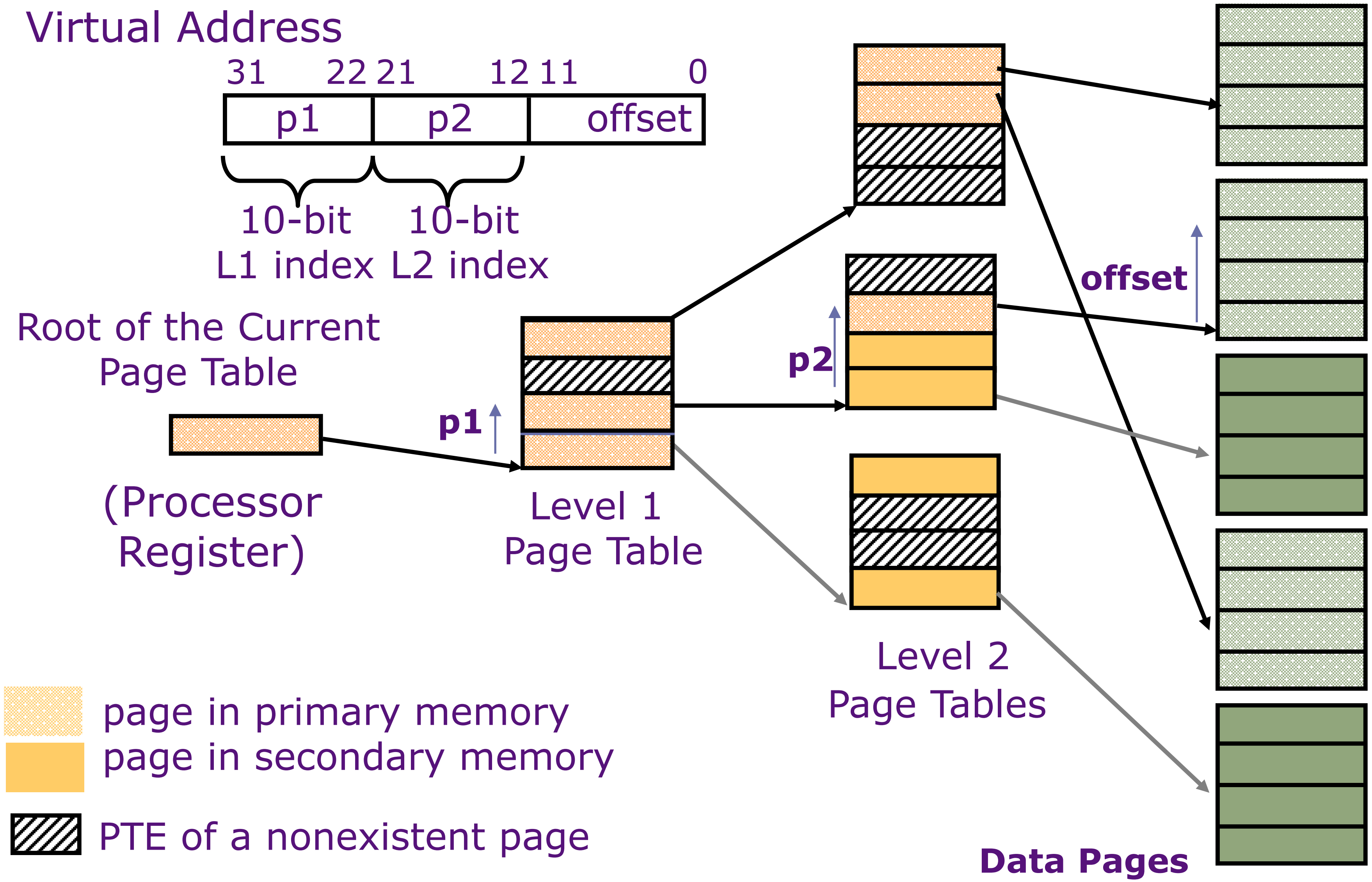
- Internal fragmentation (Not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

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



Virtual Address

31 22 21 12 11 0

p1 | p2 | offset

10-bit L1 index

10-bit L2 index

Root of the Current Page Table

(Processor Register)

p1

Level 1 Page Table

p2

Level 2 Page Tables

offset

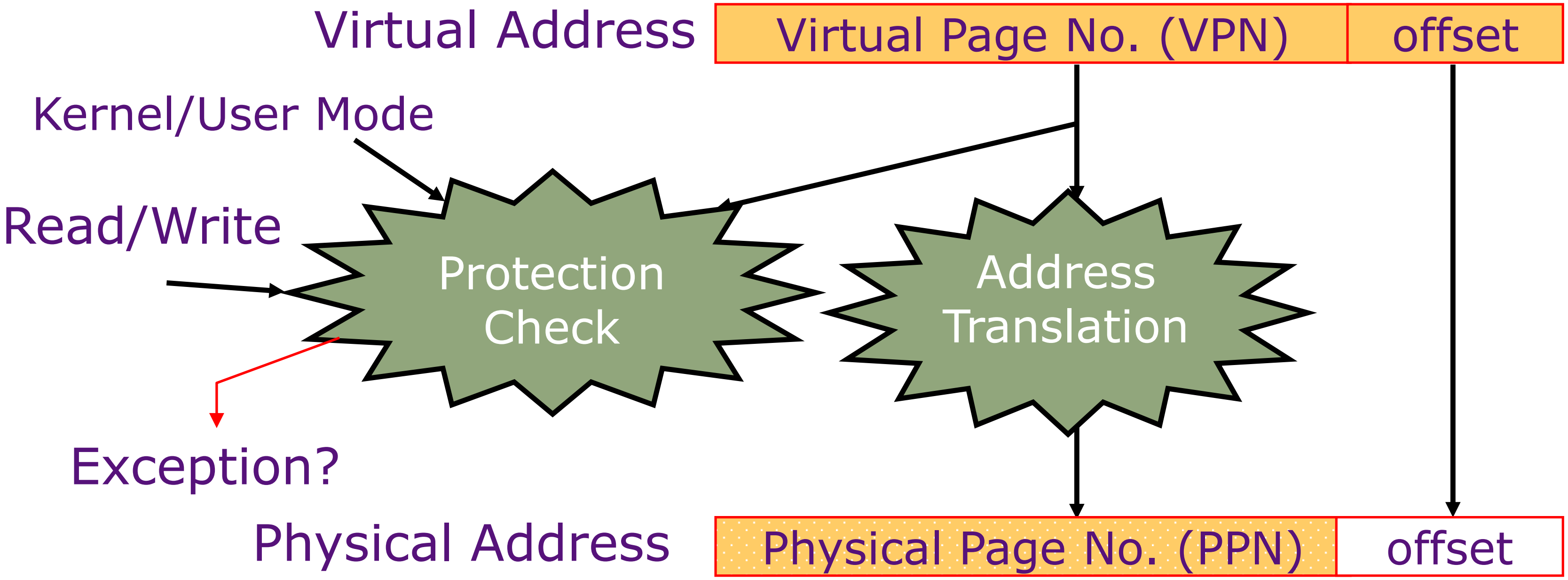
Data Pages

░ page in primary memory

🟨 page in secondary memory

▨ PTE of a nonexistent page

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

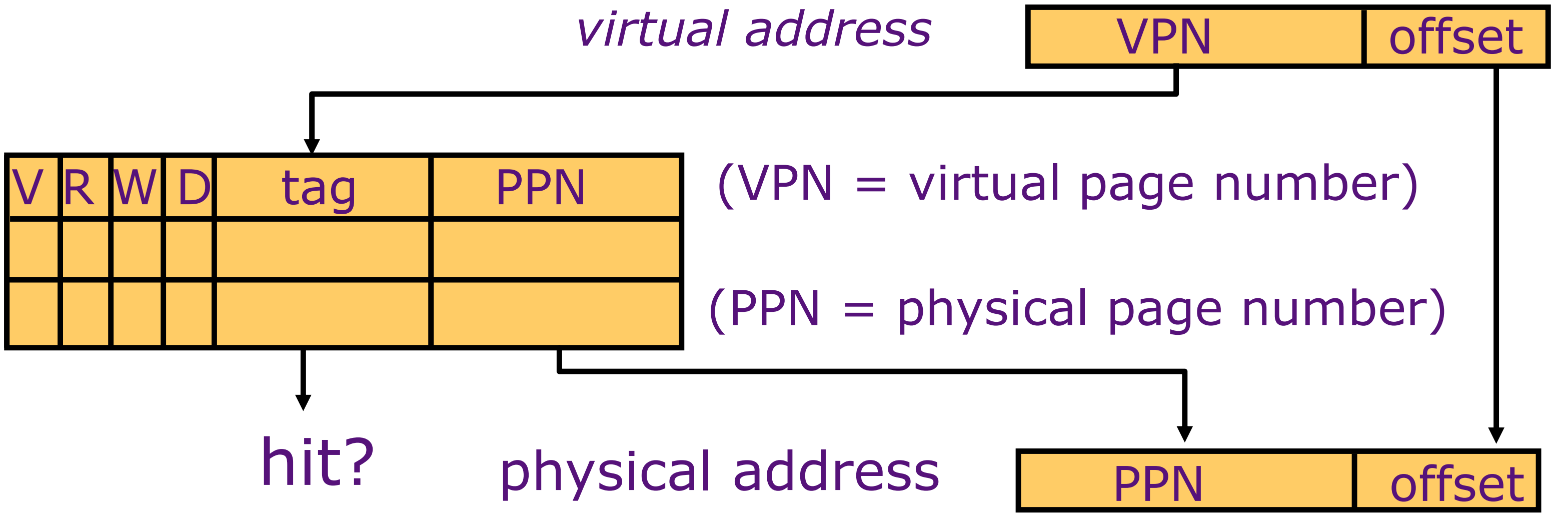
Address translation is very expensive!

In a hierarchical 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*



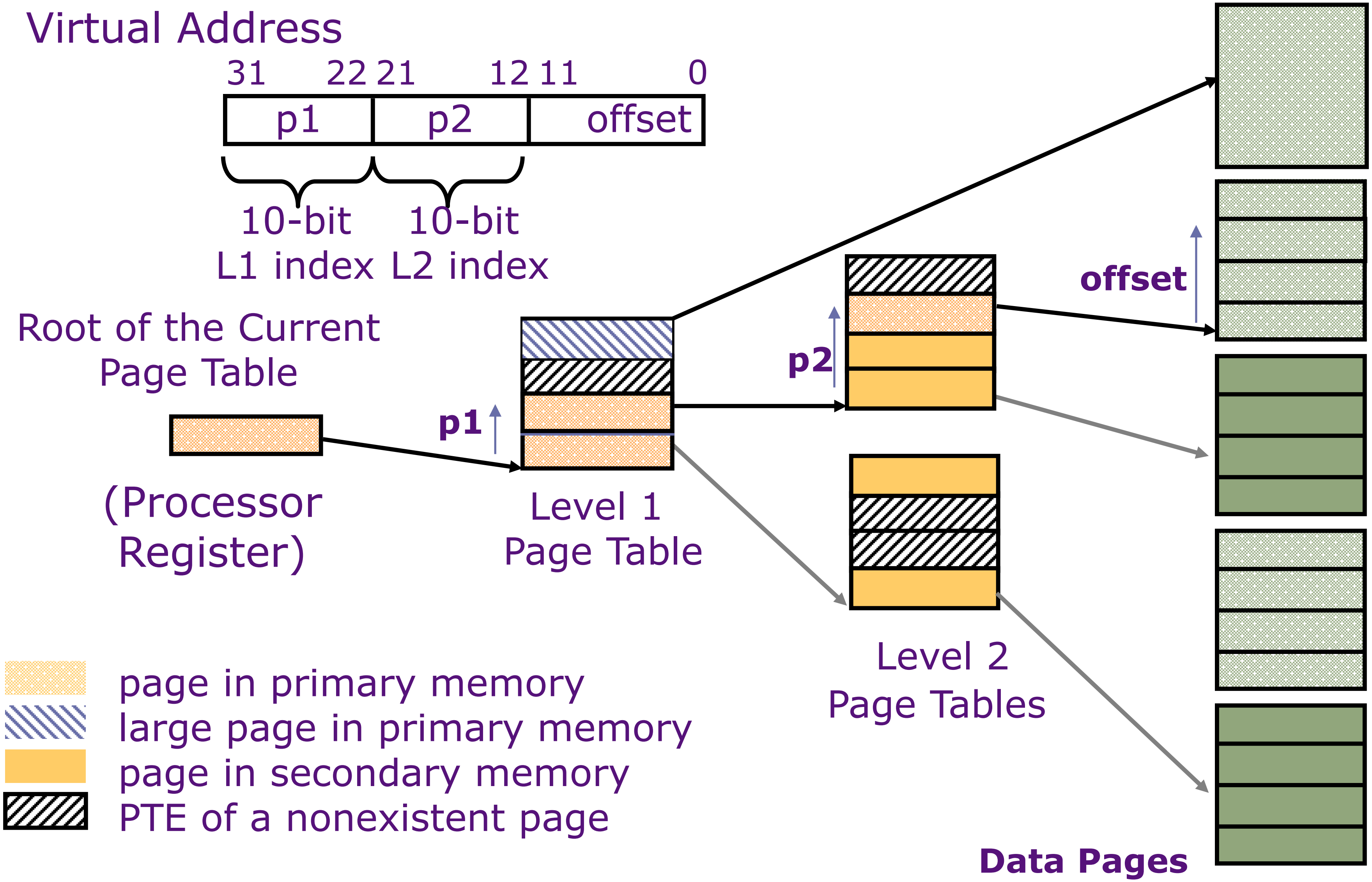
TLB Designs

- Typically 32-128 entries, usually highly associative
- Keep process information in TLB?
  - No process id → Must flush on context switch
  - Tag each entry with process id → No flush, but costlier
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB

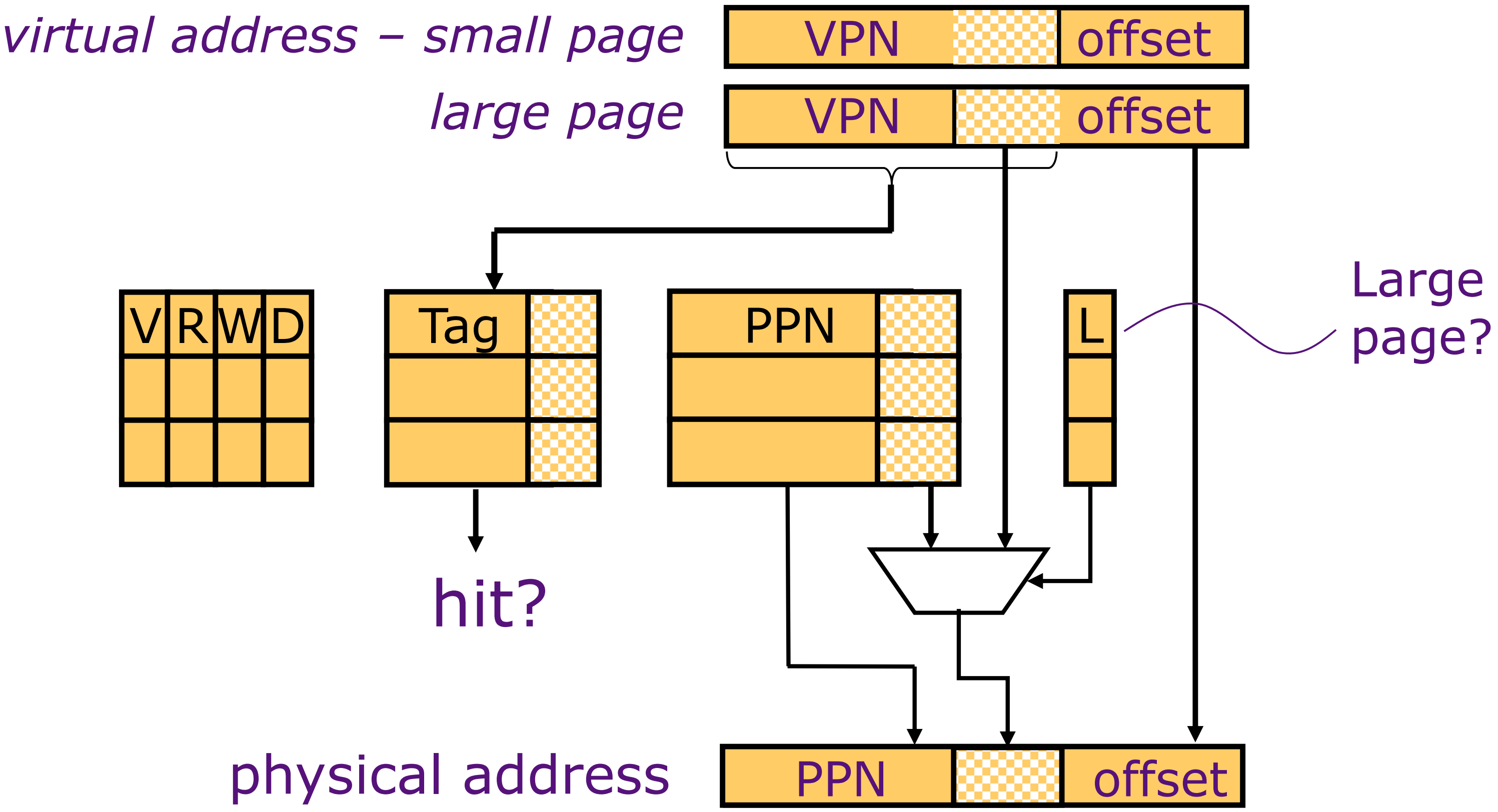
  Example: 64 TLB entries, 4KB pages, one page per entry

  TLB Reach = *64 entries * 4 KB = 256 KB (if contiguous)* ?
- Ways to increase TLB reach
  - Multi-level TLBs (e.g., Intel Skylake: 64-entry L1 data TLB, 128-entry L1 instruction TLB, 1.5K-entry L2 TLB)
  - Multiple page sizes (e.g., x86-64: 4KB, 2MB, 1GB)

Variable-Sized Page Support



Variable-Size Page TLB



Alternatively, have a separate TLB for each page size (pros/cons?)

Handling a TLB Miss

Software (MIPS, Alpha)

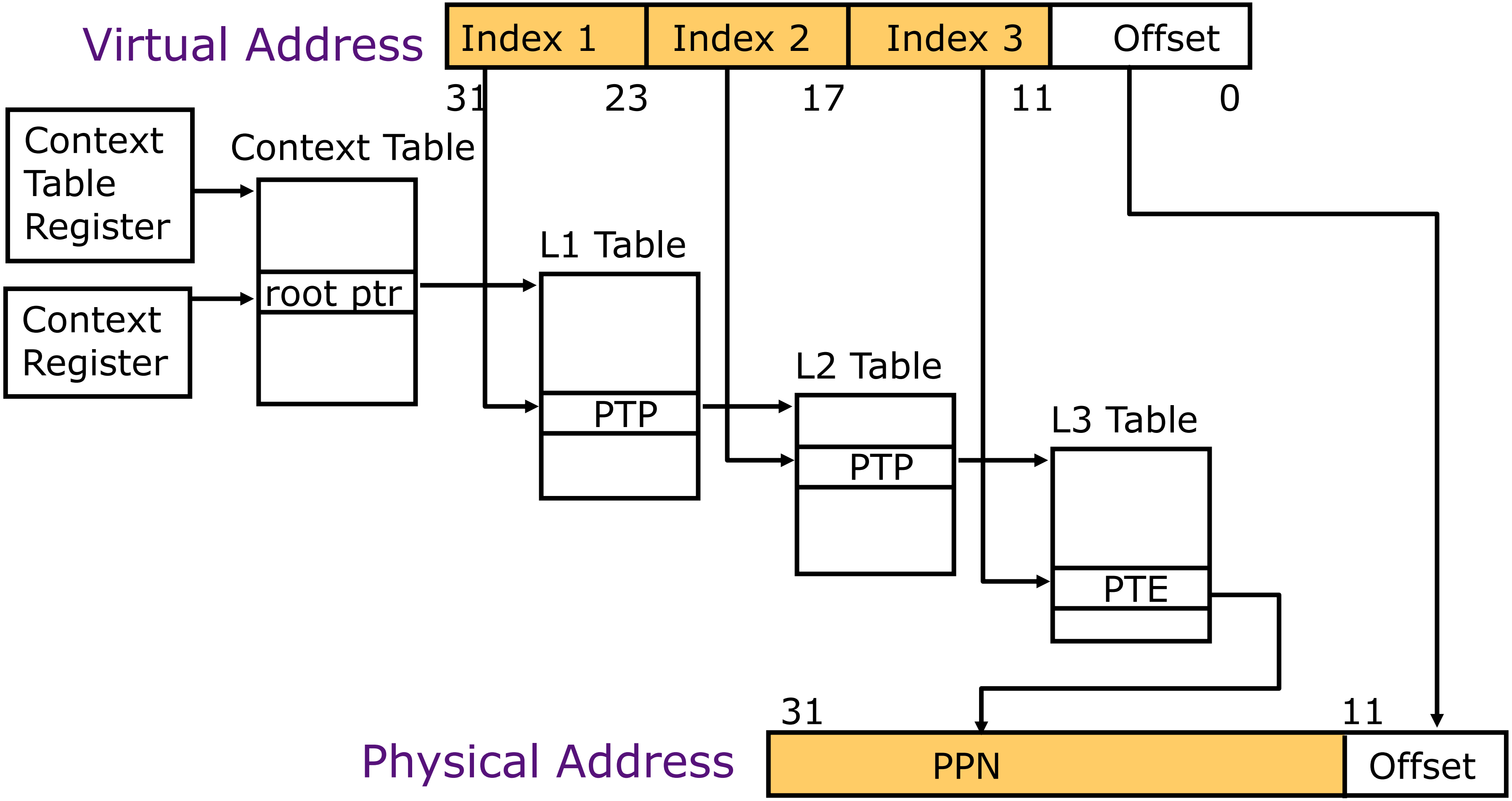
TLB miss causes an exception and the operating system walks the page tables and reloads TLB. *A privileged "untranslated" addressing mode used for walk*

Hardware (SPARC v8, x86, PowerPC)

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

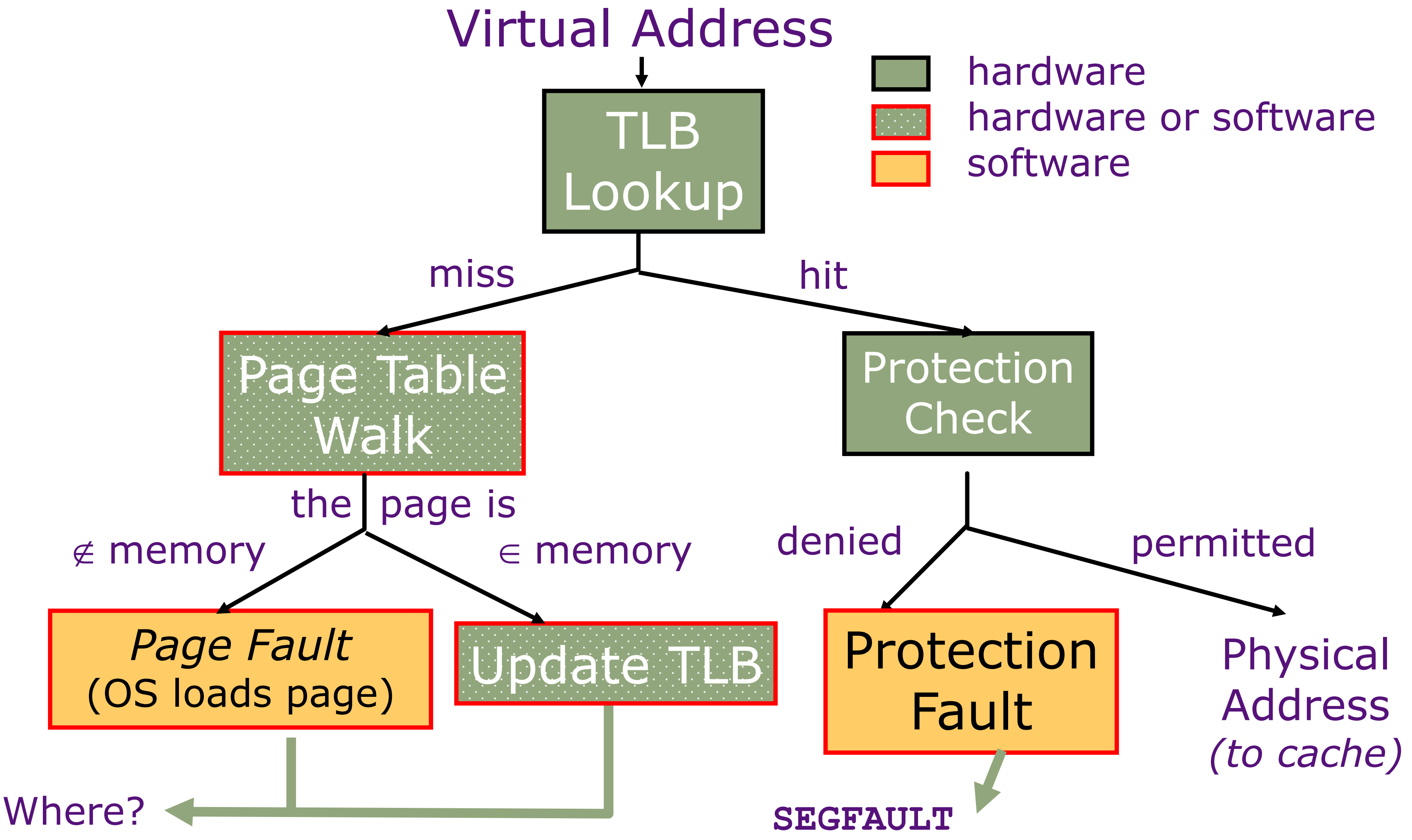
If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page-Fault exception for the original instruction

Hierarchical Page Table Walk: SPARC v8



MMU does this table walk in hardware on a TLB miss

Address Translation:
putting it all together



Virtual Address

🟩 hardware
▨ hardware or software
🟧 software

TLB Lookup

miss

hit

Page Table Walk

Protection Check

the page is

∉ memory

∈ memory

denied

permitted

Page Fault (OS loads page)

Update TLB

Protection Fault

Physical Address *(to cache)*

Where?

SEGFAULT

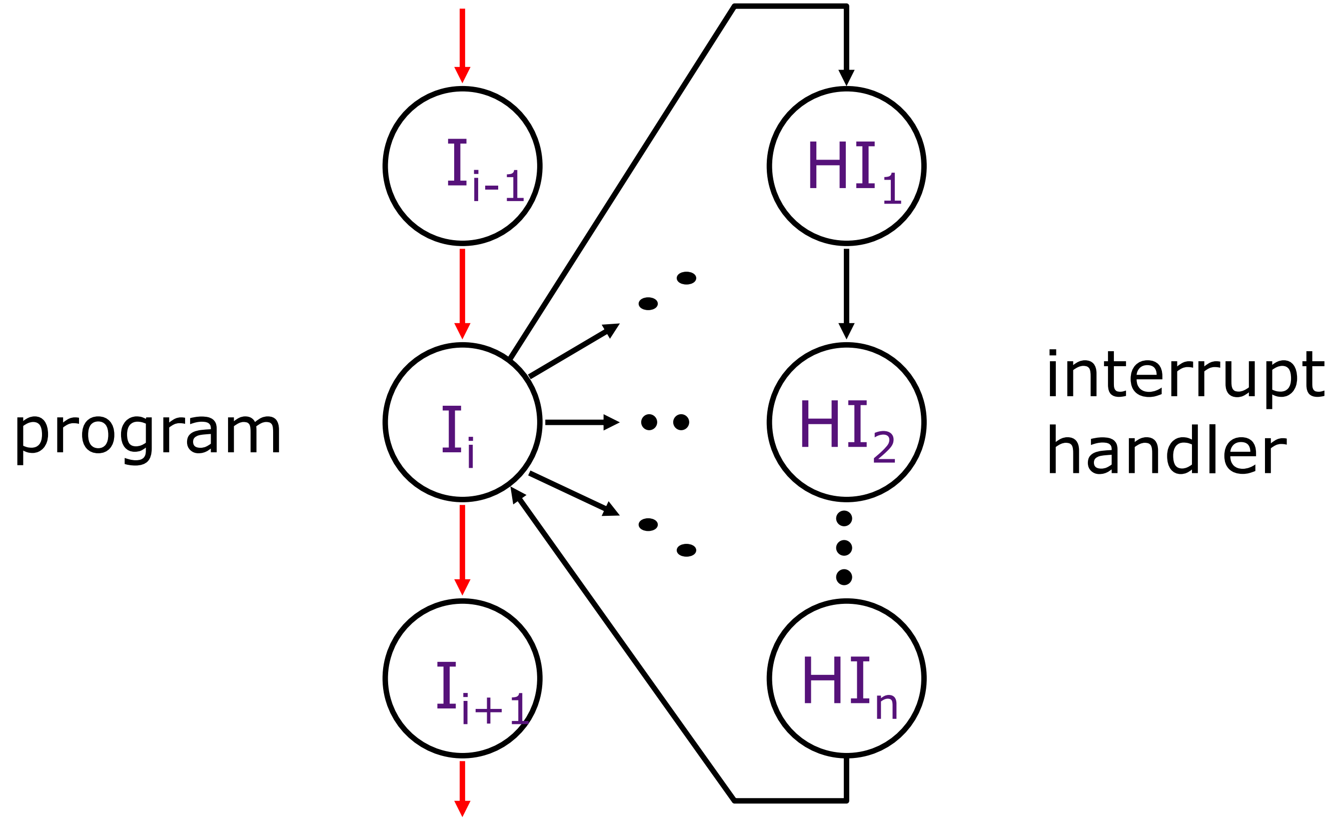
Topics

- Interrupts

- Speeding up the common case:
  - TLB & Cache organization

- Modern Usage

Interrupts:
altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program’s point of view.

Causes of Interrupts

Interrupt: an *event* that requests the attention of the processor

- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. exception)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception
  - misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations
  - *traps:* system calls, e.g., jumps into kernel

Asynchronous Interrupts

Invoking the interrupt handler

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- When the processor decides to process interrupt
  - It stops the current program at instruction I_i, completing all the instructions up to I_{i-1} (*precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
  - It disables interrupts and transfers control to a designated interrupt handler running in kernel mode

Interrupt Handler

- Saves EPC before enabling interrupts to allow nested interrupts ⇒
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read a *status register* that indicates the cause of the interrupt
- Uses a special indirect jump instruction RFE (*return-from-exception*) that
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state

Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a *particular instruction*
- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - With pipelining, requires undoing the effect of one or more partially executed instructions
- In case of a trap (system call), the instruction is considered to have been completed
  - A special jump instruction involving a change to privileged kernel mode

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
- 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

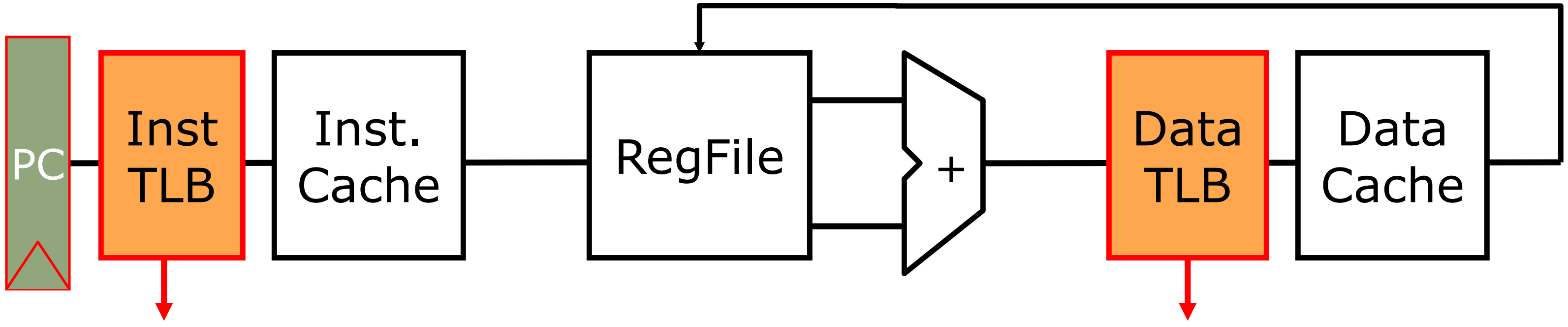
Topics

- Interrupts

- Speeding up the common case:
  - TLB & Cache organization

- Modern Usage

Address Translation in CPU

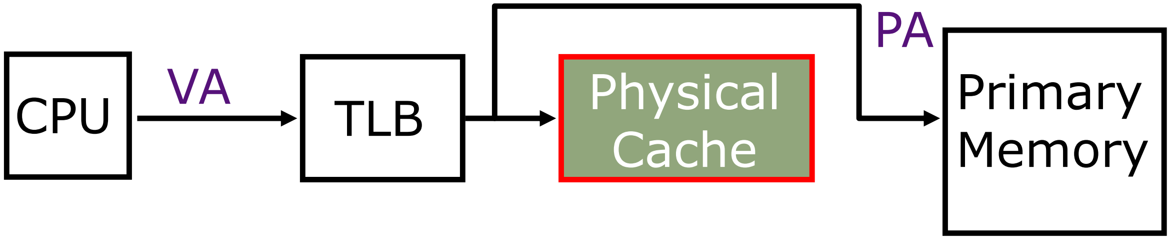


TLB miss? Page Fault? Protection violation?

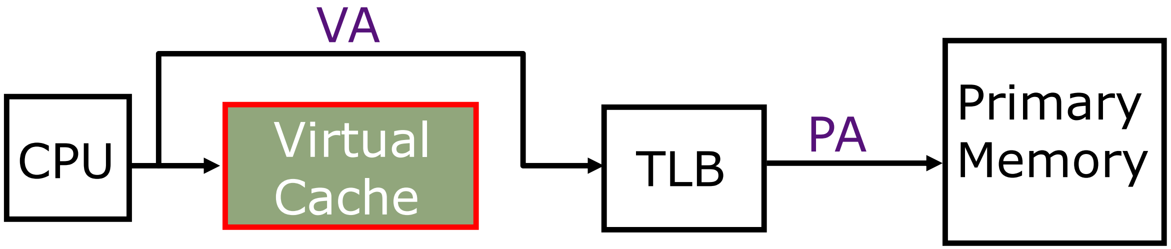
TLB miss? Page Fault? Protection violation?

- Software handlers need a *restartable* exception on page fault or protection violation
- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Need mechanisms to cope with the additional latency of TLB:
  - slow down the clock
  - pipeline the TLB and cache access
  - virtual-address caches
  - parallel TLB/cache access

Virtual-Address Caches

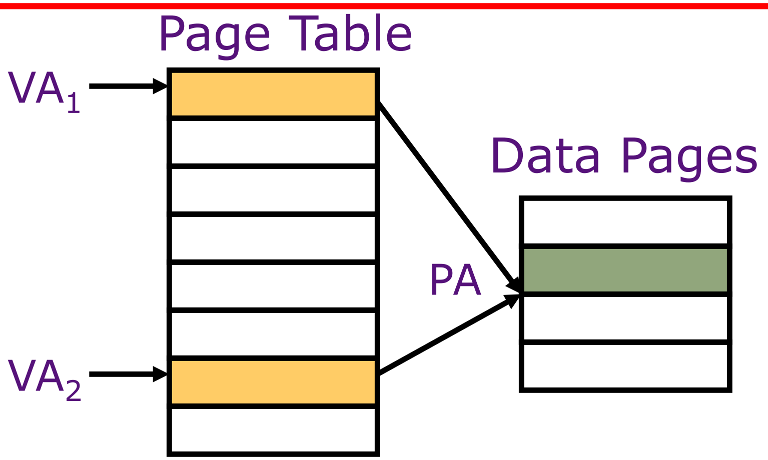


Alternative: place the cache before the TLB



- 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 (-)

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

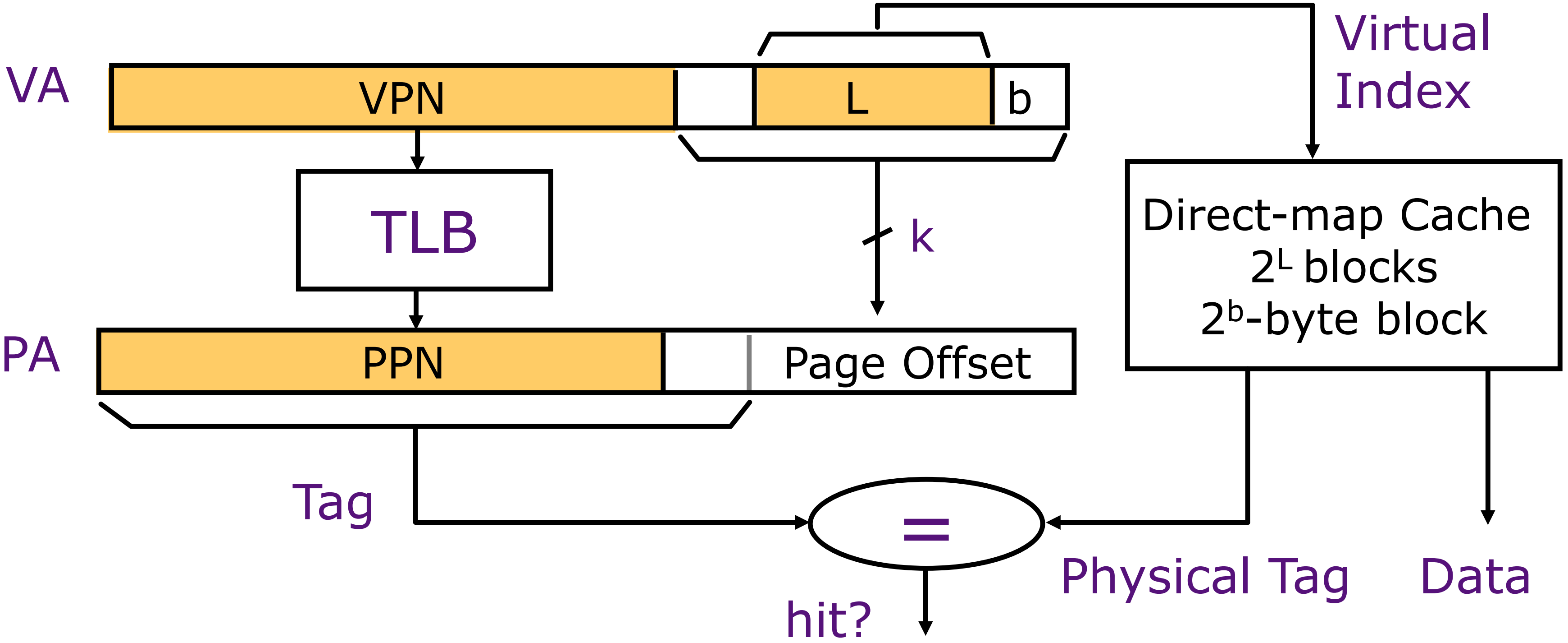
Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist 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



Index L is available without consulting the TLB

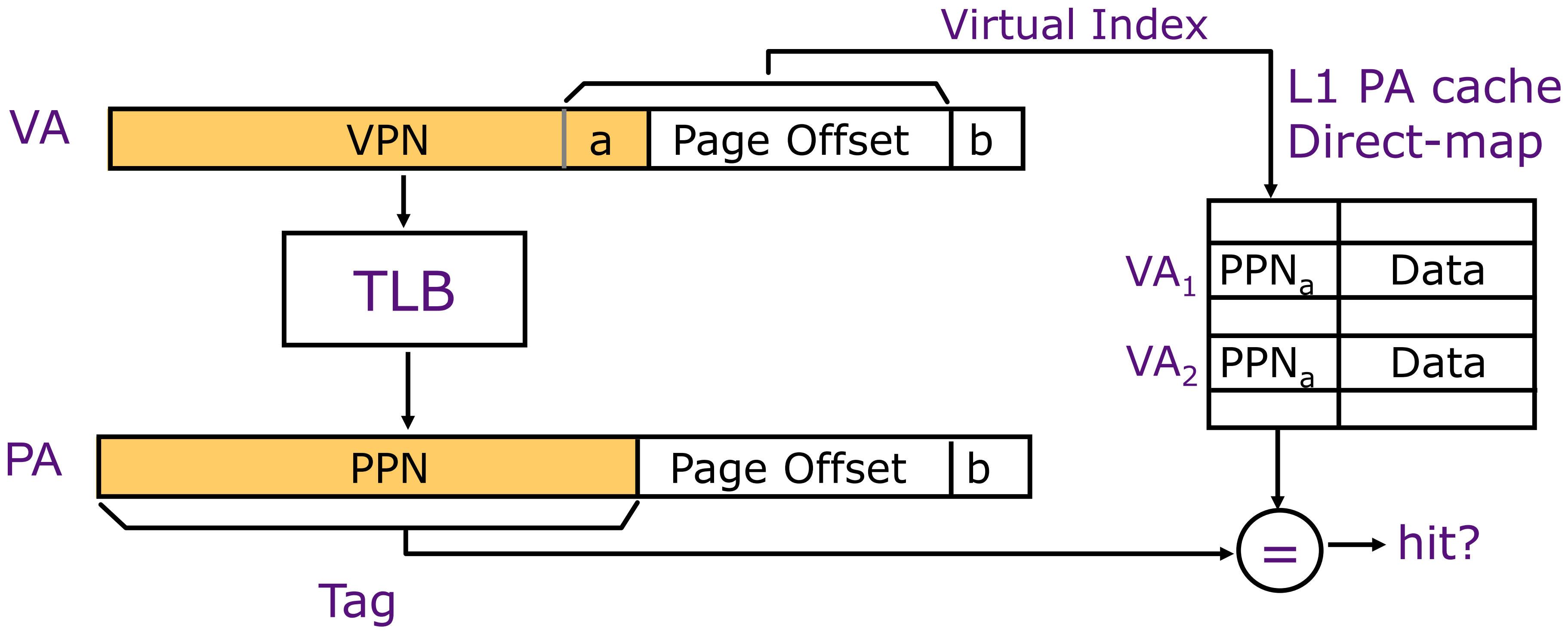
⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

When does this work? $L + b < k$ ✓ $L + b = k$ ✓ $L + b > k$ ✗

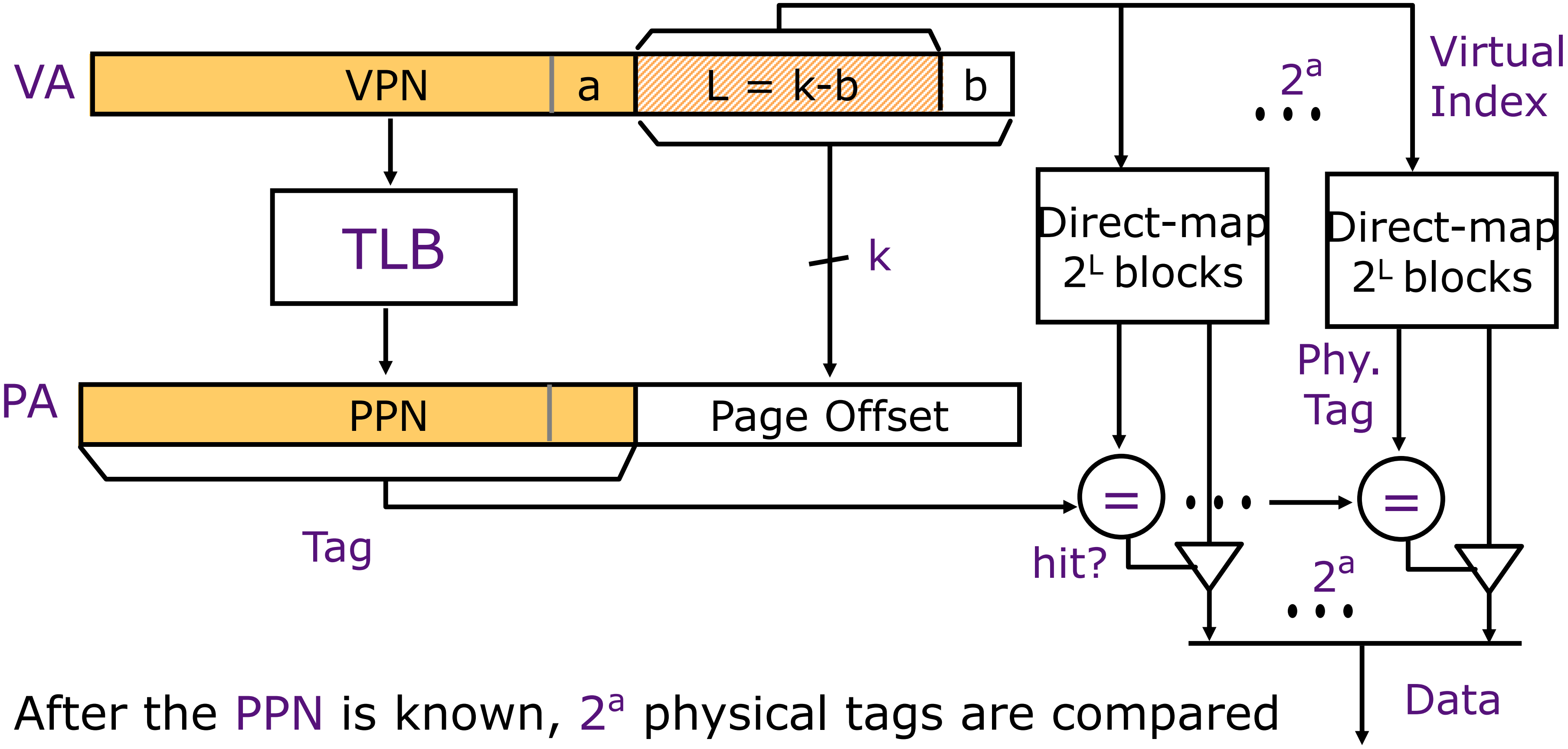
Concurrent Access to TLB & Large L1

The problem with L1 > Page size



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

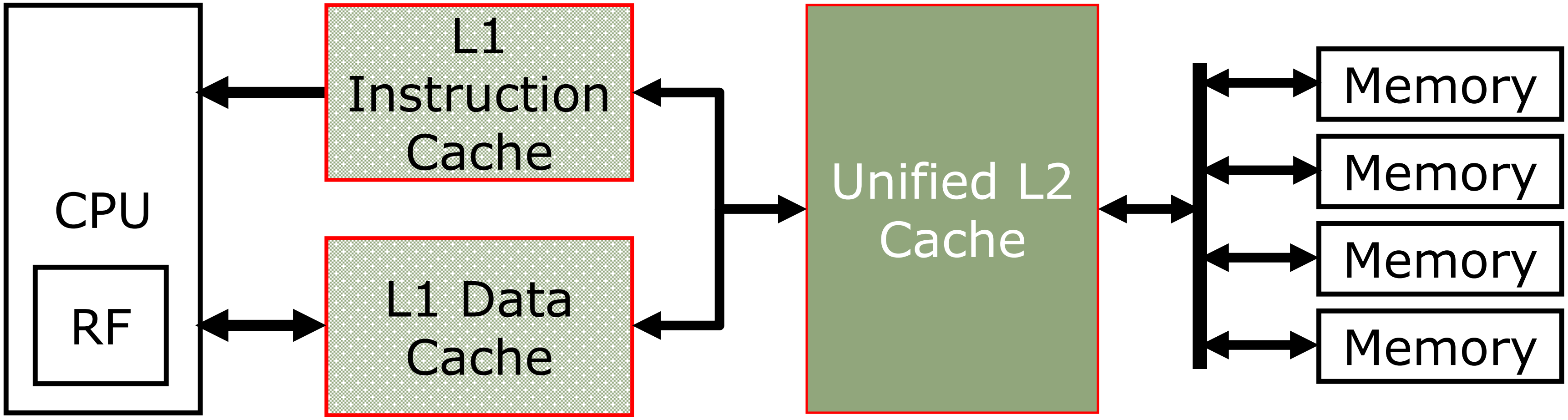
Virtual-Index Physical-Tag Caches:
Associative Organization



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

Is this scheme realistic?

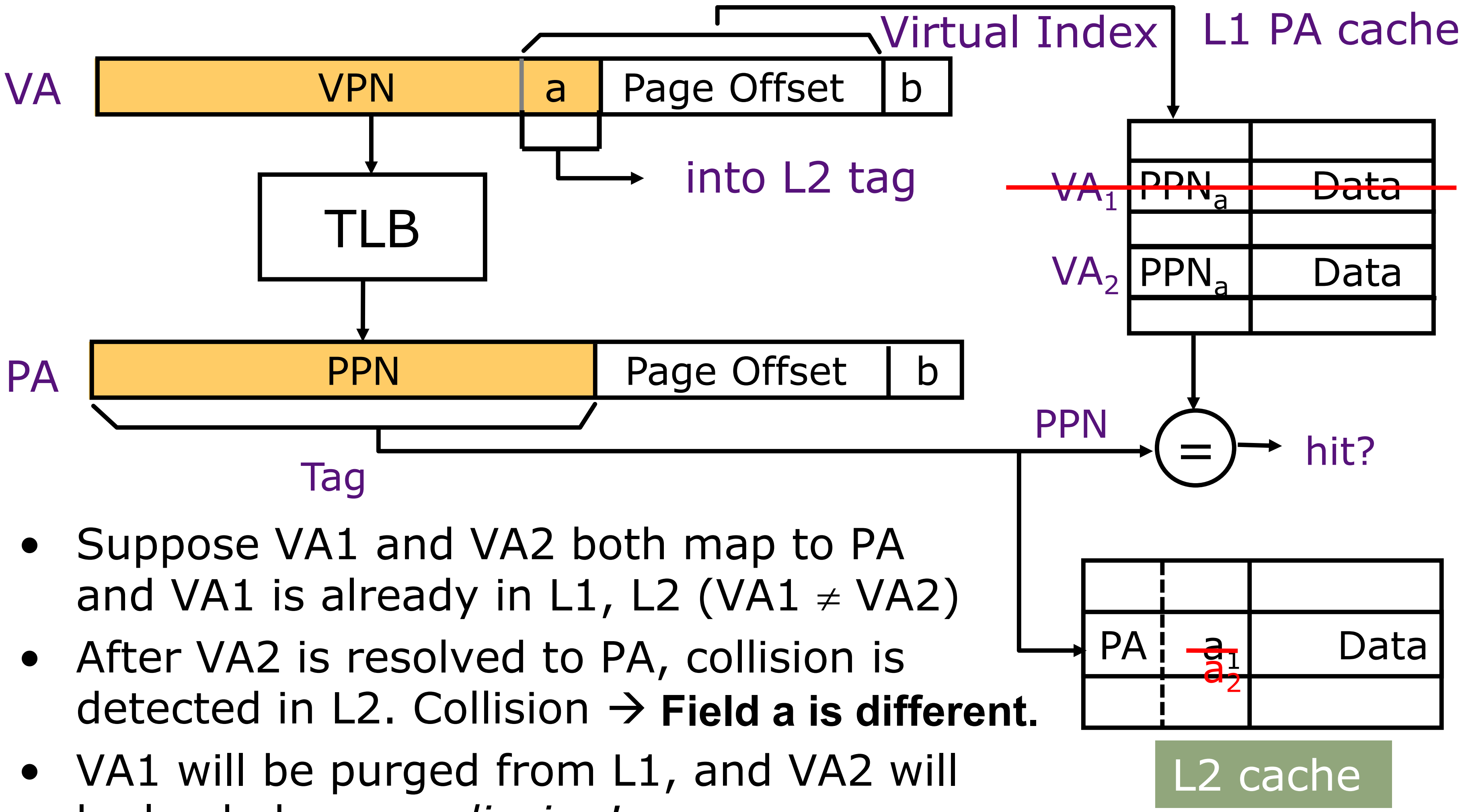
A solution via Second-Level Cache



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

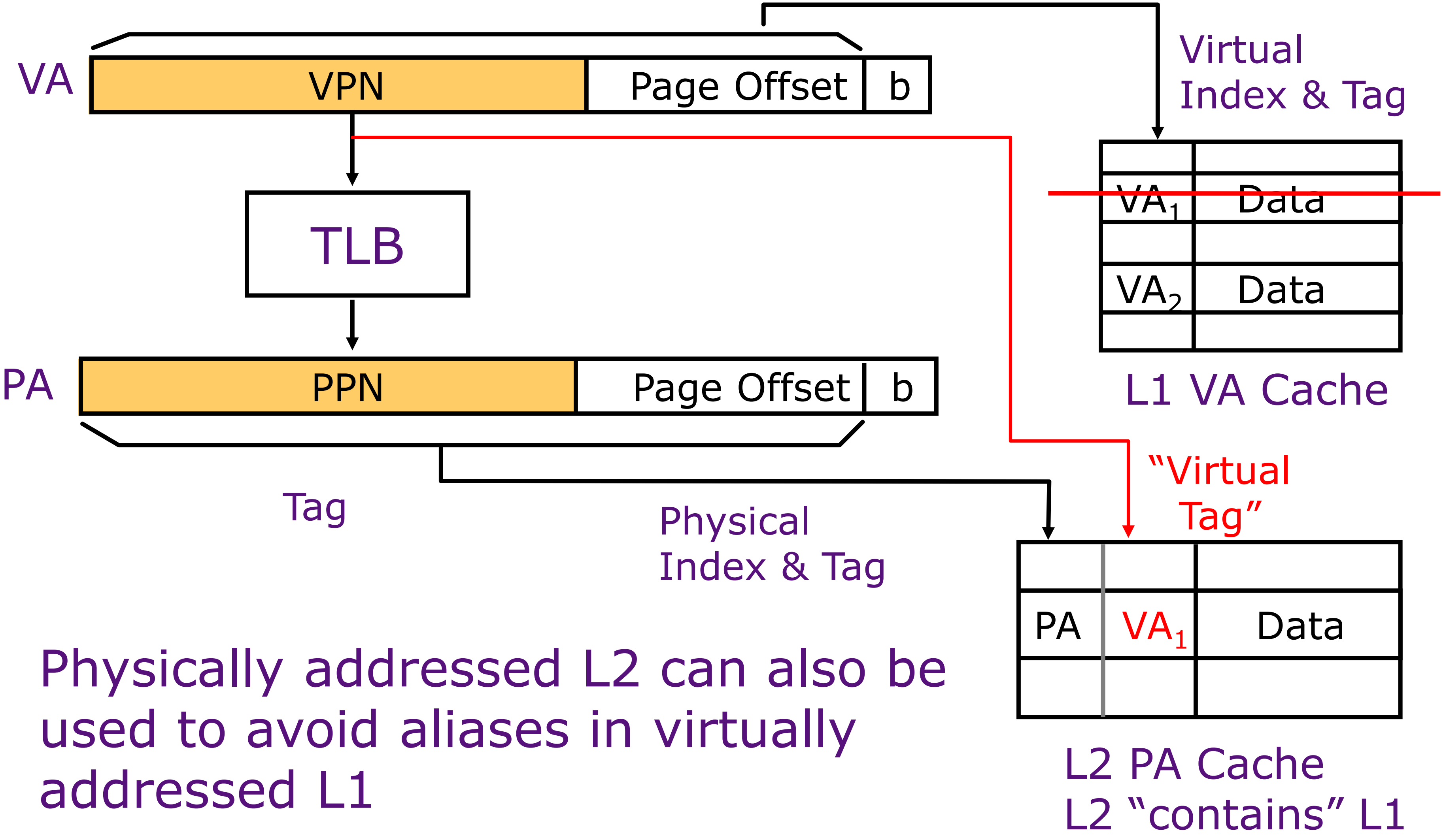
L2 is "inclusive" of both Instruction and Data caches

Anti-Aliasing Using L2: *MIPS R10000*



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

Virtually Addressed L1:
Anti-Aliasing using L2

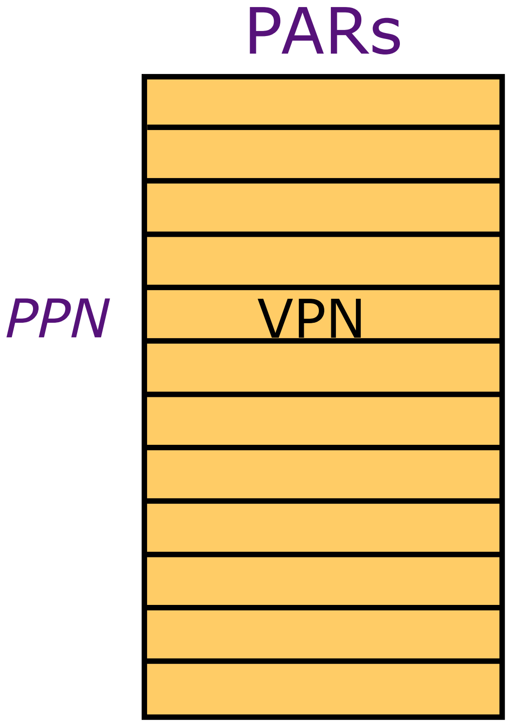


Physically addressed L2 can also be used to avoid aliases in virtually addressed L1

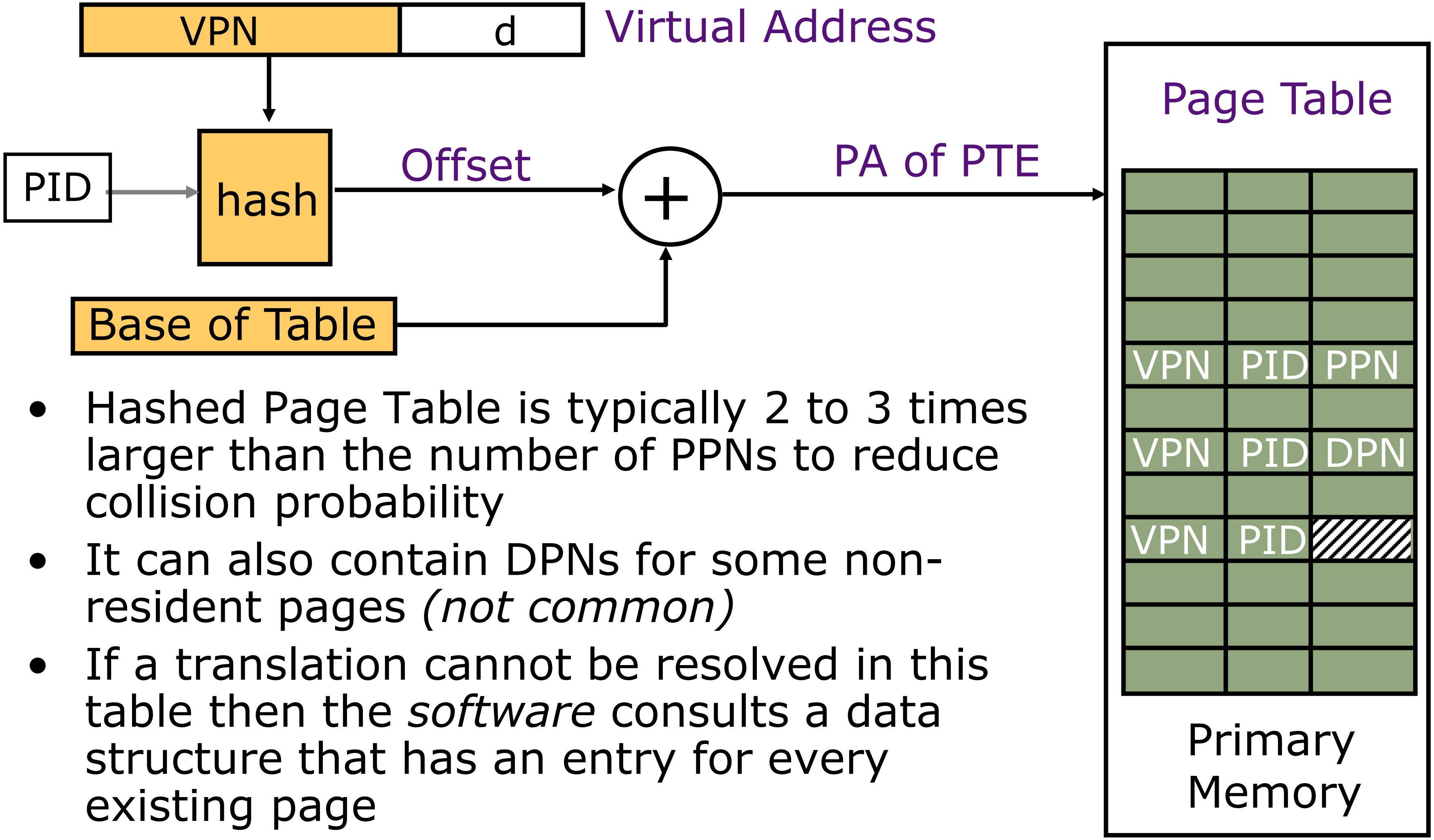
Atlas Revisited

- 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?*

  Must check all PARs!



Hashed Page Table:
Approximating Associative Addressing



- Hashed Page Table is typically 2 to 3 times larger than the number of PPNs to reduce collision probability
- It can also contain DPNs 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

Topics

- Interrupts
- Speeding up the common case:
  - TLB & Cache organization
- Modern Usage

Virtual Memory Use Today - 1

- Desktop/server/cellphone processors 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 and GPUs have translation and protection but not demand paging
  (Older Crays: base&bound, Japanese & Cray X1: pages)
  - Don't waste expensive processor 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

Next lecture: Pipelining!