

**CSE 431**  
**Computer Architecture**  
**Fall 2022**  
**Exploiting the Memory Hierarchy: TLBs**

Kiwan Maeng

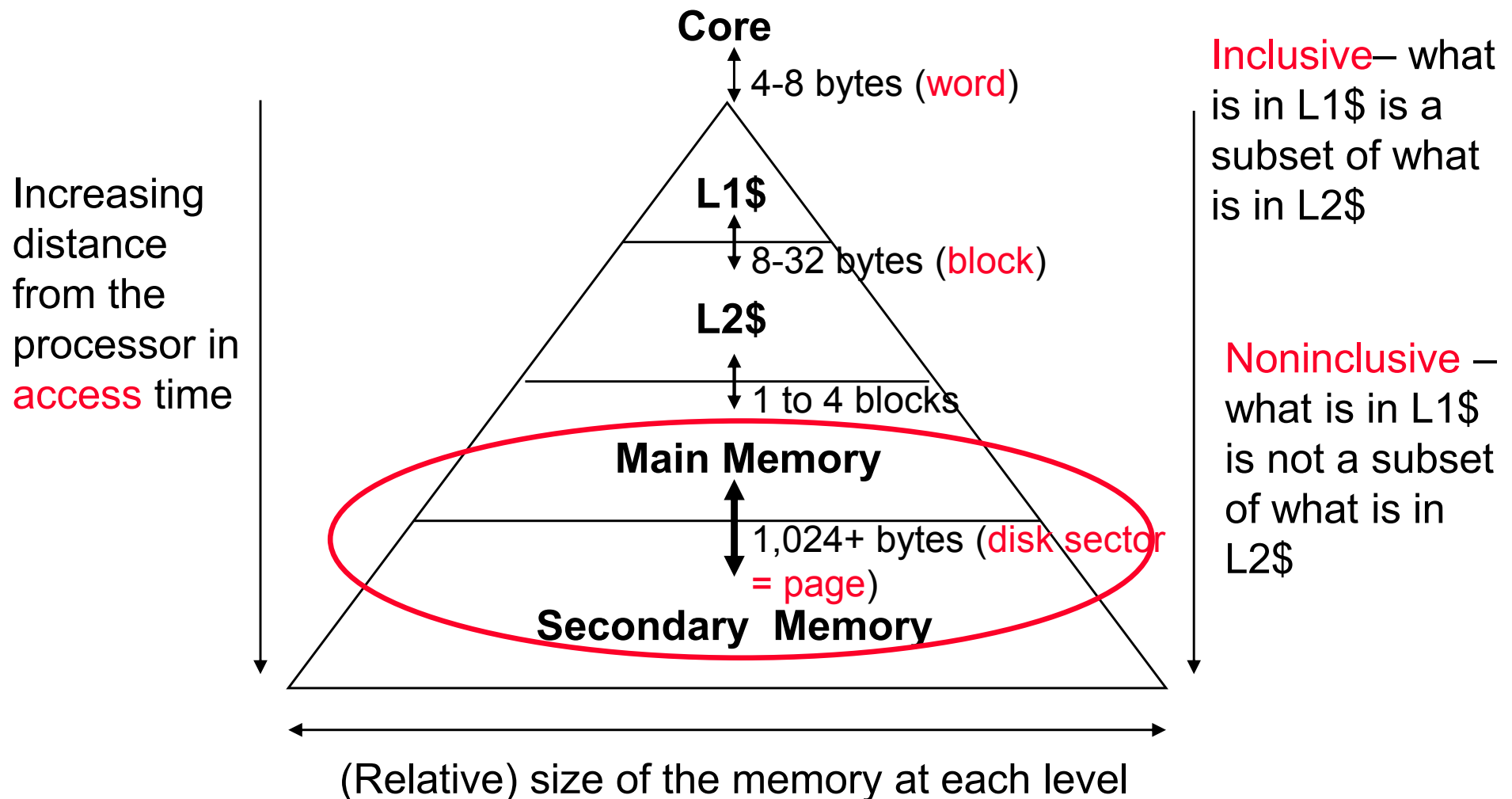
[Adapted from *Computer Organization and Design, 5<sup>th</sup> Edition*,  
Patterson & Hennessy, © 2014, MK]

# Review: Caches

- ❑ Caches exploit temporal and spatial locality
  - ❑ Convert “data reuse” into “data locality”
- ❑ Managed by hardware
- ❑ Built typically as a hierarchy (e.g., L1-L2-L3)
- ❑ Hardware optimizations include flexible placement of data, prefetching, careful hierarchy design
- ❑ Software optimizations include computation reordering and data layout restructuring

# Review: The Memory Hierarchy


- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



# How is the Hierarchy Managed?

- ❑ registers  $\leftrightarrow$  memory
  - ❑ by compiler (programmer?)
- ❑ registers  $\leftrightarrow$  cache  $\leftrightarrow$  main memory
  - ❑ by the cache controller hardware
- ❑ main memory  $\leftrightarrow$  secondary memory (flash, disk)
  - ❑ by the operating system (virtual memory)
    - virtual address to physical address mapping
    - assisted by the hardware (TLB, page tables)
  - ❑ by the programmer with OS support (files)

# Virtual Memory Concepts

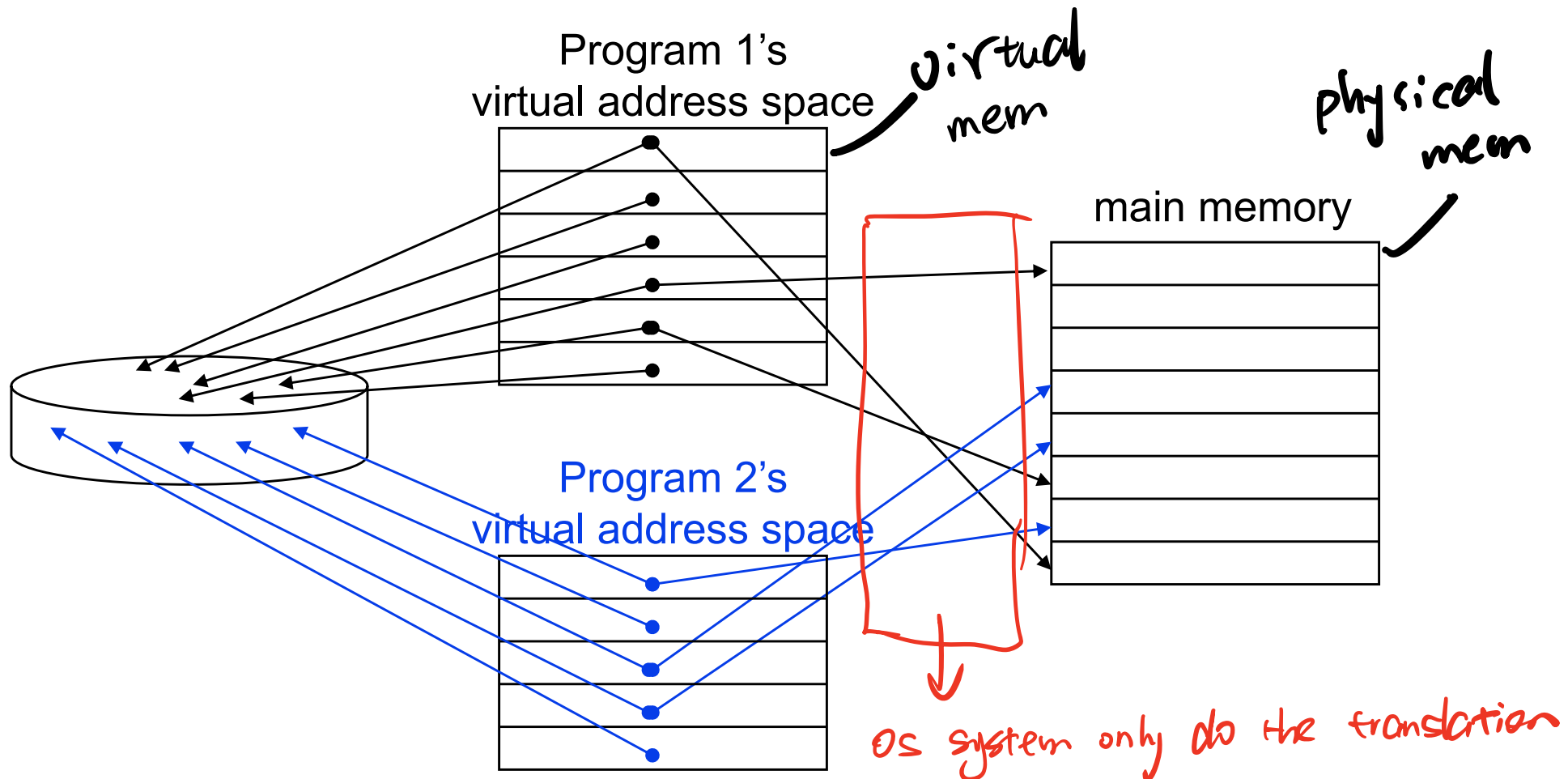
- ❑ Use main memory as a “cache” for secondary memory
  - ❑ Allows efficient and **safe** sharing of main memory among multiple processes/threads (running programs)
    - Each program is compiled into its own **private virtual address space**
  - ❑ Provides the ability to run programs and data sets larger than the size of physical memory
  - ❑ Simplifies loading a program for execution by providing for **code relocation** (i.e., the code/data can be loaded in main memory anywhere the OS can find space for it)
- ❑ The core and OS work together to translate virtual addresses to physical addresses
  - ❑ A virtual memory miss (i.e., when the page is not in physical memory) is called a **page fault** 
- ❑ What makes it work efficiently? – **the Principle of Locality**
  - ❑ Programs tend to access only a small portion of their address space over long portions of their execution time

# Virtual Memory Concepts

- ❑ Use main memory as a “cache” for secondary memory
  - ❑ Allows efficient and **safe** sharing of main memory among multiple processes/threads (running programs)
    - Each program is compiled into its own **private virtual address space**
  - ❑ Provides the ability to run programs and data sets larger than the size of physical memory
  - ❑ Simplifies loading a program for execution by providing for **code relocation** (i.e., the code/data can be loaded in main memory anywhere the OS can find space for it)
- ❑ The core and OS work together to translate virtual addresses to physical addresses
  - ❑ A virtual memory miss (i.e., when the page is not in physical memory) is called a **page fault**
- ❑ What makes it work efficiently? – **the Principle of Locality**
  - ❑ Programs tend to access only a small portion of their address space over long portions of their execution time

# Two Programs Sharing Physical Memory

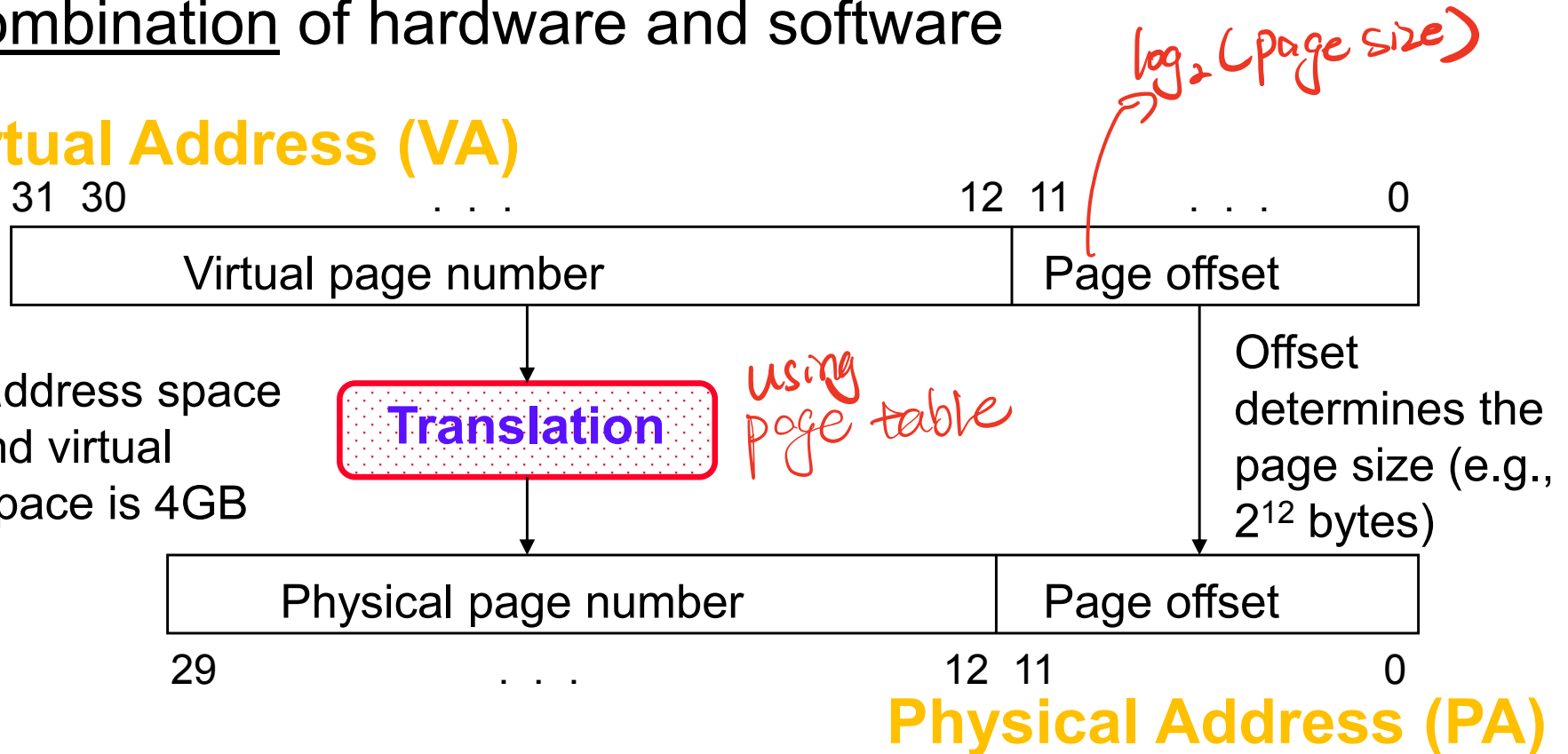
- ❑ A program's address space is divided into **pages** (all one fixed size) or segments (variable sizes)
  - ❑ The starting location of each page (either in main memory or in secondary memory) is contained in the program's **page table**



# Address Translation

- A **virtual address** is translated to a **physical address** by a combination of hardware and software

## Virtual Address (VA)



- So, each memory request *first* requires an **address translation** from the virtual space to the physical space





# Virtual Address Translation Mechanisms

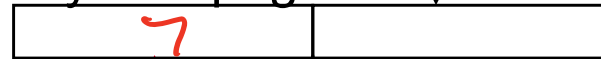
Virtual page #    Offset



Virtual Address

The page table together with the program counter and the registers specifies the **state** of a program

Physical page #

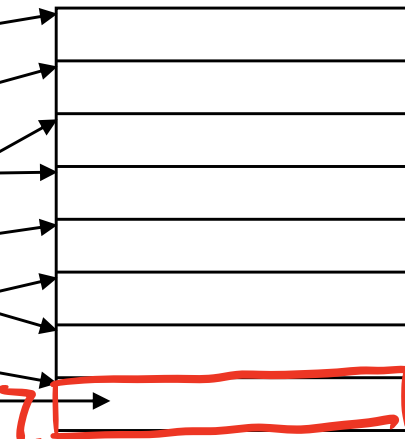


Physical Address

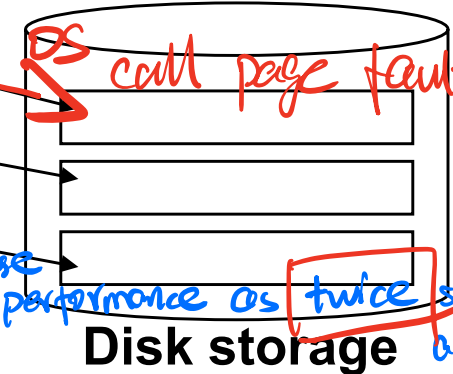
Page Table Register

Physical page base addr

V	Physical page base addr
1	•
1	•
1	•
1	•
1	•
1	•
1	•
0	•
1	•
0	•
1	•
0	•



Main memory



Disk storage

Page Table

(stored in main memory)

note that page table is in main memory. so every time we need to access

page table and then the main mem will cause the performance as

twice

slow as usual

such that TLB needed ↑

If **valid bit** for a virtual page is off (0), a page fault occurs

0, 1 represent page fault or not

call page fault

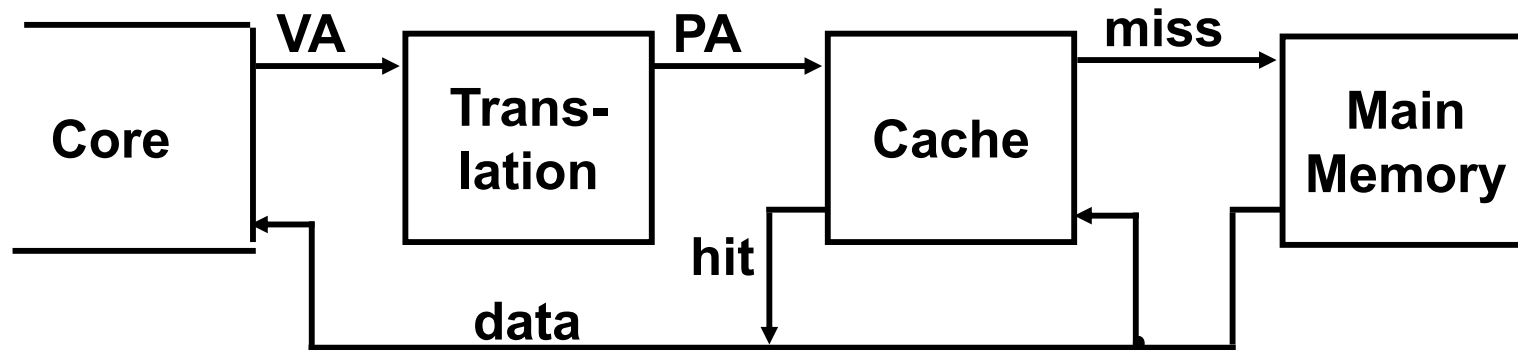
# Design Choices in Virtual Memory Systems

→ 4kB

- ❑ Pages should be large enough to amortize high access times
- ❑ Organizations that reduce page fault rate (e.g., full associative placement of pages in main memory) are attractive
- ❑ Page faults can be handled in software
- ❑ Write-through will not work

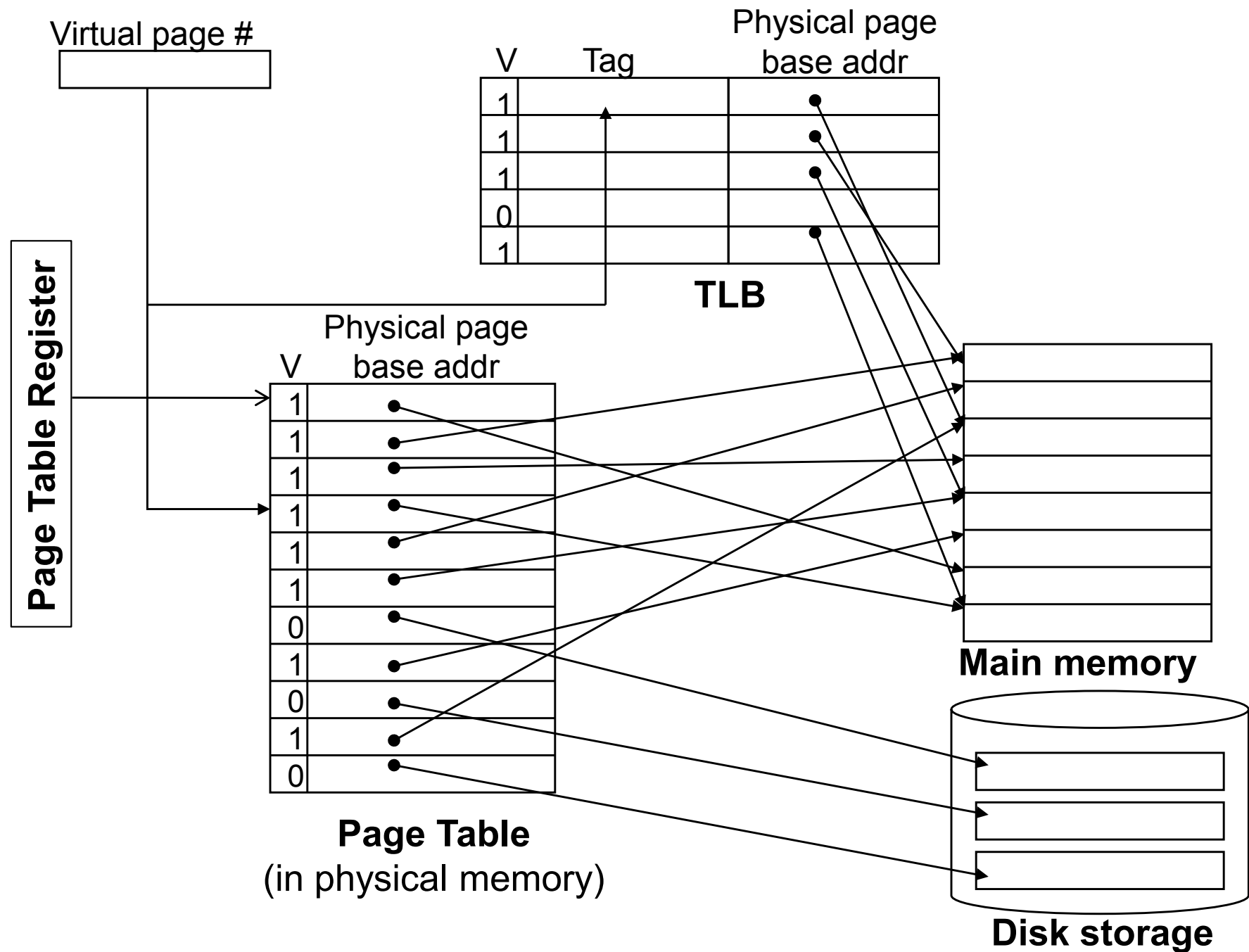
# Virtual Addressing with a Cache

- ❑ Thus, it takes an *extra* memory access to translate a VA to a PA



- ❑ This makes memory (cache) accesses **very expensive** (if every access is really *two* accesses)
- ❑ The hardware fix is to use a **Translation Lookaside Buffer** (TLB) – a fast, small **cache** that keeps track of recently used **address mappings** to avoid having to do a page table lookup in memory (i.e., cache or main memory)
- ❑ Typical TLBs – 16 to 512 PTEs, 0.5 to 2 cycle for a hit, 10-100 cycles for a miss, 0.01% to 1% miss rate

# Making Address Translation Fast



# Translation Lookaside Buffers (TLBs)

*usually full associative TLB*

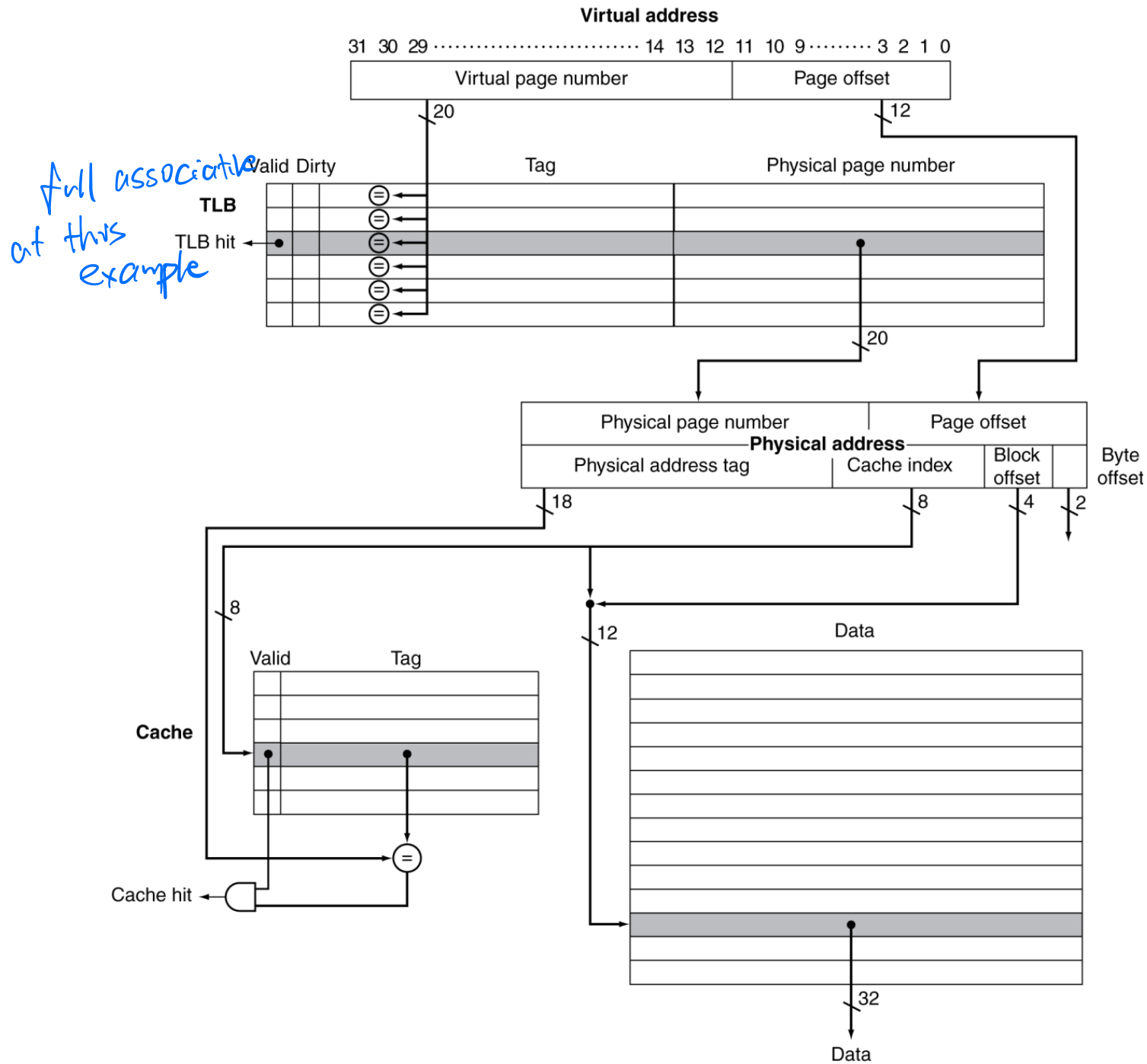
- Just like any other cache, the TLB can be organized as *fully associative*, *set associative*, or *direct mapped*
  - `simplescalar` defaults are `itbl:16:4096:4:1` (16 sets per way, 4-way set associative so 64 entries, 4096B pages) and `dtlb:32:4096:4:1` and `tlb:lat 30` (cycles to service a TLB miss)

V	Virtual Page #	Physical Page #	Dirty	Ref	Access

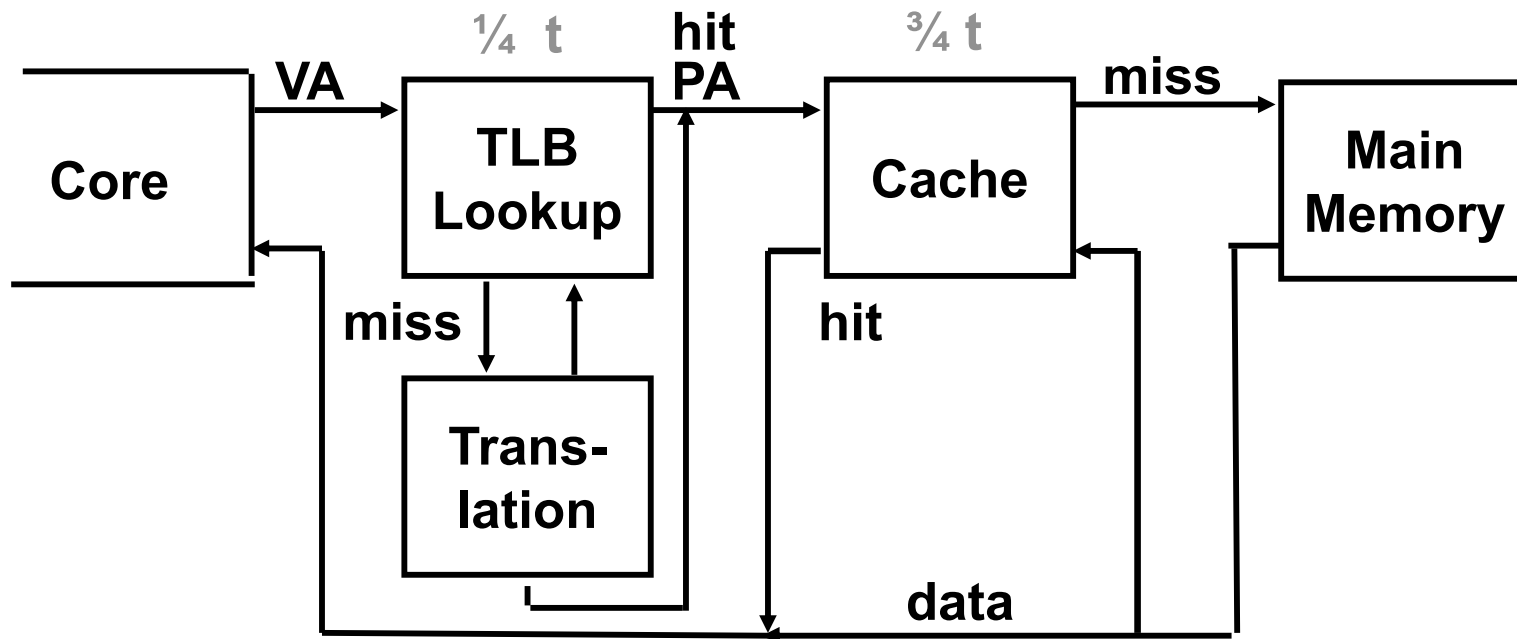
V = Valid?, Dirty = is the page dirty (so will have to be written back on replacement)?, Ref = Referenced recently?, Access = Write access allowed?

- TLB access time is typically much smaller than cache access time (because TLBs are *much smaller* than caches)
  - TLBs are typically not more than 512 entries even on high end machines

# TLB with Cache Example



# A TLB in the Memory Hierarchy



## ❑ A TLB miss – is it a page fault or merely a TLB miss?

- ❑ If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
  - Takes 10's of cycles to find and load the translation info into the TLB
- ❑ If the page is not in main memory, then it's a *true* page fault
  - Takes 1,000,000's of cycles to service a page fault
  - Page faults can be handled in software because the overhead will be small compared to the disk access time

## ❑ TLB misses are much more frequent than true page faults

# TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – this is what we want!
Hit	Hit	Miss	Yes – although the page table is not checked after the TLB hits
Miss	Hit	Hit	Yes – TLB missed, but PA is in page table and data is in cache; <u>update TLB</u>
Miss	Hit	Miss	Yes – TLB missed, but PA is in page table, data not in cache; <u>update TLB</u>
Miss	Miss	Miss	Yes – page fault; OS takes control
Hit	Miss	Miss/ Hit	No – TLB translation is not possible if the page is not present in main memory
Miss	Miss	Hit	No – data is not allowed in the cache if the page is not in memory

when TLB hits, we will not go to page table



# Handling a TLB Miss

- ❑ A **TLB miss** can indicate one of two possibilities:
  - ❑ A page is present in memory, and we need only create the missing TLB entry
  - ❑ A page is not present in memory, and we need to transfer control to the operating system to deal with a page fault
- ❑ MIPS traditionally handles a TLB miss in *software*
- ❑ Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process
- ❑ A TLB miss or page fault exception must be asserted by the end of the same clock cycle that the memory access occurs, so that the next clock cycle will begin exception processing rather than continue normal instruction execution

# Handling a <sup>page fault</sup> TLB Miss

- ❑ Once the operating system knows the virtual address that caused the page fault, it must complete three steps:
  - ❑ Look up the page table entry using the virtual address and find the location of the referenced page on disk
  - ❑ Choose a physical page to replace; if the chosen page is dirty, it must be written out to disk before we can bring a new virtual page into this physical page
  - ❑ Start a read to bring the referenced page from disk into the chosen physical page
- ❑ The last step will take *millions of clock cycles* (so will the second if the replaced page is dirty)
- ❑ Accordingly, the operating system will usually select/schedule another process to execute in the processor *until* the disk access completes – *context switching*
- ❑ When the read of the page from the disk completes, the operating system can restore the state of the process that originally caused the page fault and execute the instruction that returns from the exception
- ❑ The user process (application) then re-executes the instruction that faulted

# Some Virtual Memory Design Parameters

	Paged VM	TLBs
Total size (blocks)	16,000 to 250,000	40 to 1,024
Total size (KB)	1,000,000 to 1,000,000,000	0.25 to 16
Block size (B)	4000 to 64,000	4 to 32
Hit time		0.25 to 1 clock cycle
Miss penalty (clocks)	10,000,000 to 100,000,000	10 to 1,000
Miss rates	0.00001% to 0.0001%	0.01% to 1%

# TLBs

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	<p>1 TLB for instructions and 1 TLB for data</p> <p>Both TLBs are fully associative, with 32 entries, round robin replacement</p> <p>TLB misses handled in hardware</p>	<p>1 TLB for instructions and 1 TLB for data per core</p> <p>Both L1 TLBs are four-way set associative, LRU replacement</p> <p>L1 I-TLB has 128 entries for small pages, 7 per thread for large pages</p> <p>L1 D-TLB has 64 entries for small pages, 32 for large pages</p> <p>The L2 TLB is four-way set associative, LRU replacement</p> <p>The L2 TLB has 512 entries</p> <p>TLB misses handled in hardware</p>

# TLB Management

## ❑ Hardware-managed TLB

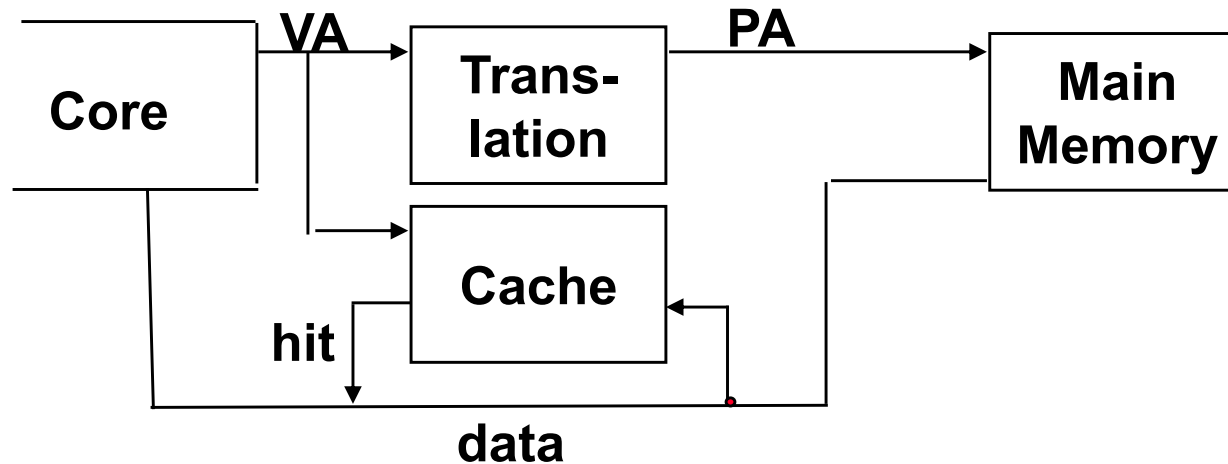
- ❑ No need for expensive interrupts
- ❑ Pipeline remains largely unaffected
- ❑ OS cannot employ alternate design

## ❑ Software-managed TLB

- ❑ Data structure design is flexible since the OS controls the page table walk
- ❑ Miss handler is also instructions
  - It may itself miss in the instruction cache
- ❑ Data cache may be polluted by the page table walk

# Why Not a Virtually Addressed Cache?

- ❑ A virtually addressed cache would only require address translation on cache misses



But,

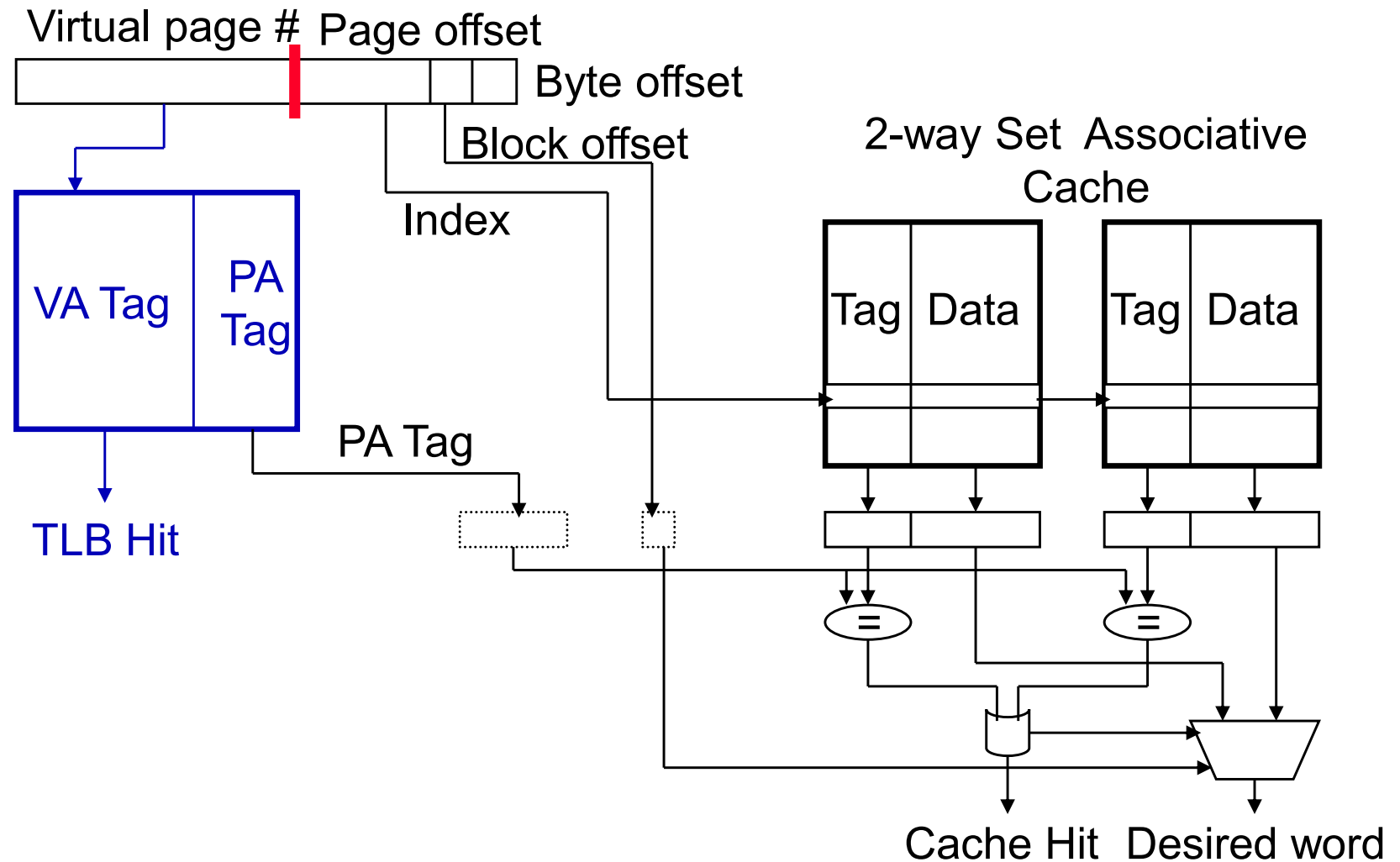
- ❑ Two programs which are sharing data will have two different virtual addresses for the same physical address – **aliasing** – so will have two copies of the shared data in the cache and two entries in the TBL which would lead to *coherence* issues
  - Must update all cache entries with the same physical address or the memory becomes inconsistent

## Possible cache organizations:

- 1) physically-indexed, physically-tagged
- 2) virtually-indexed, virtually-tagged
- 3) virtually-indexed, physically-tagged

# Further Reducing Translation Time

- ❑ Can **overlap** the cache access with the TLB access
- ❑ Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation
- ❑ This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache



# The Hardware/Software Interface

- ❑ What parts of the virtual to physical address translation is done by or assisted by the hardware?
  - ❑ Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May need to allot an extra stage in the pipeline for TLB access
  - ❑ Page table storage, fault detection and updating
    - Page faults result in interrupts (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables
  - ❑ Disk placement
    - Bootstrap (e.g., out of disk sector 0) so the system can service a limited number of page faults before the OS is even loaded



# Memory and OS Protection

- ❑ Different processes can share parts of their virtual address spaces
  - ❑ For example, run a program under control of a debugger, or sharing data and computing effort between processes
  - ❑ Need to protect against improper access
  - ❑ Requires OS assistance via the page tables and locking mechanisms
- ❑ Hardware support for OS protection
  - ❑ Privileged supervisor mode (aka kernel mode)
  - ❑ Privileged instructions, registers and addresses
  - ❑ Page tables and other state information only accessible in supervisor mode
  - ❑ System call exception (e.g., syscall in MIPS)
  - ❑ etc.
- ❑ Important points
  - ❑ A user process can access only the storage provided by the operating system
  - ❑ Address translations, page table entries, and TLB entries can only be changed/updated by the OS
  - ❑ Controlled sharing of pages among processes can be implemented by the help of the OS and access bits in the page table that indicate whether the user program has read or write access to a page

# Common Memory Hierarchy Framework

# Common Memory Framework

## ❑ The Principle of Locality:

- ❑ Program likely to access a relatively small portion of the address space at any instant of time.
  - **Temporal Locality**: Locality in Time
  - **Spatial Locality**: Locality in Space

## ❑ Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions

1. Where can an entry be placed in the upper level?
2. How is an entry found if it is in the upper level?
3. What entry is replaced on miss?
4. How are writes handled?

## ❑ Page tables map virtual address to physical address

- ❑ TLBs are important for fast translation

## Direct vs Associate Caching Memory Organizations

## One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

## Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

### Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

### Eight-way set associative (fully associative)

[illegible]

## Q1&Q2: Where can a entry be placed/found?

	# of sets	Entries per set
Direct mapped	# of entries	1
Set associative	(# of entries)/ associativity	Associativity (typically 2 to 16)
Fully associative	1	# of entries

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all entries' tags Separate lookup (page) table	# of entries 0

### Q3: Which entry should be replaced on a miss?

- ❑ Easy for direct mapped – only one choice
- ❑ Set associative or fully associative
  - ❑ Random
  - ❑ LRU (Least Recently Used)
- ❑ For a 2-way set associative, random replacement has a miss rate about 10% higher than LRU
- ❑ LRU is too costly to implement for high levels of associativity ( $> 16$ -way) since tracking the usage information is costly

## Q4: What happens on a write?

- ❑ Write-through – The information is written to the entry in the current memory level *and* to the entry in the next level of the memory hierarchy
  - ❑ Always combined with a write buffer so write waits to next level memory can be eliminated (as long as the write buffer doesn't fill)
- ❑ Write-back – The information is written only to the entry in the current memory level. The modified entry is written to next level of memory only when it is replaced.
  - ❑ Need a dirty bit to keep track of whether the entry is clean or dirty
  - ❑ Virtual memory systems always use write-back of dirty pages to disk
- ❑ Pros and cons of each?
  - ❑ Write-through: read misses don't result in writes (so are simpler and cheaper), easier to implement
  - ❑ Write-back: writes run at the speed of the cache; repeated writes require only one write to lower level

# Handling writes (stores)

## ❑ Do we allocate a cache block on a write miss?

- ❑ Allocate on write miss: Yes
- ❑ No-allocate on write miss: No

## ❑ Allocate on write miss

- + Can consolidate writes instead of writing each of them individually to next level
- + Simpler because write misses can be treated the same way as read misses
- Requires (?) transfer of the whole cache block

## ❑ No-allocate

- + Conserves cache space if locality of writes is low (potentially better cache hit rate)



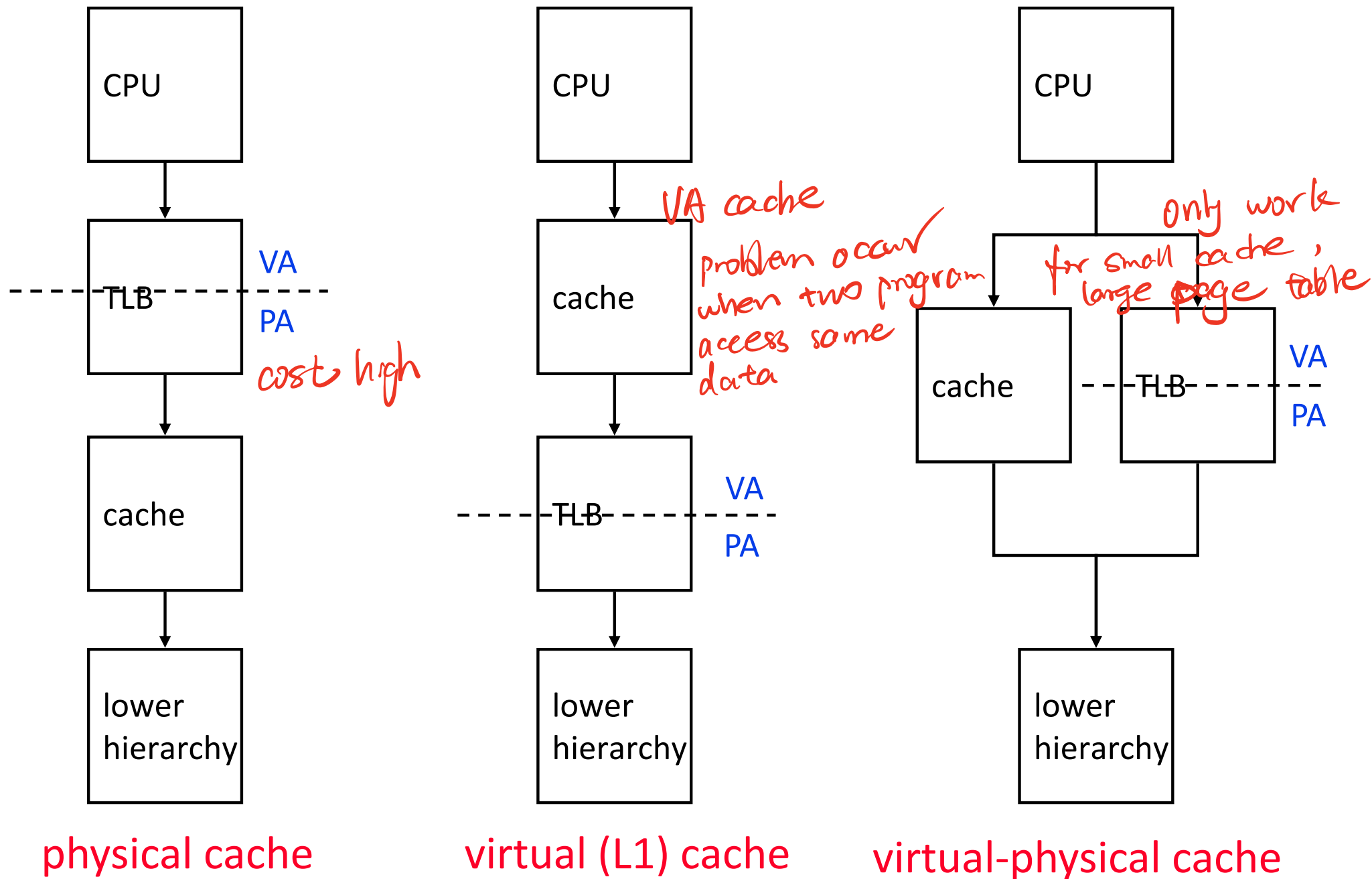
# Memory hierarchy design challenges

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

# Cache versus page replacement

- ❑ Physical memory (DRAM) is a cache for disk
  - ❑ Usually managed by system software via the virtual memory subsystem
- ❑ Page replacement is similar to cache replacement
- ❑ Page table is the “tag store” for physical memory data store
- ❑ What is the difference?
  - cache ❑ Hardware versus software — page table
  - ❑ Number of blocks in a cache versus physical memory
  - ❑ “Tolerable” amount of time to find a replacement candidate

# Cache-virtual memory interaction



# Virtually-Indexed Physically-Tagged

- ❑ If  $C \leq (\text{page\_size} \times \text{associativity})$ , the cache index bits come only from page offset (same in VA and PA)
- ❑ If both cache and TLB are on chip
  - ❑ index both arrays concurrently using VA bits
  - ❑ check cache tag (physical) against TLB output at the end

