CMSC 611: Advanced Computer Architecture

Cache & Memory

Most slides adapted from David Patterson. Some from Mohomed Younis

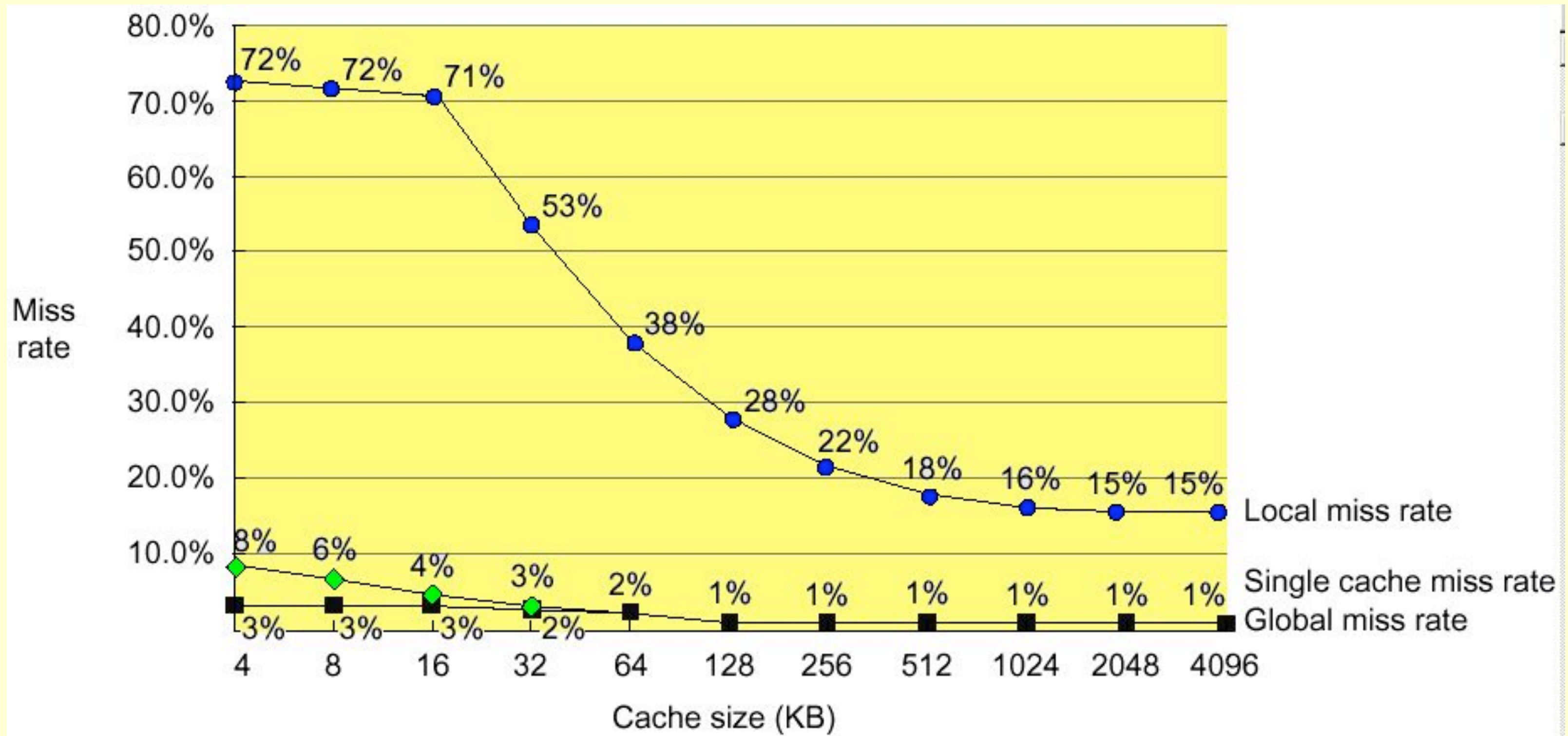
Second Level Cache

- The previous techniques reduce the impact of the miss penalty on the CPU
  - L2 cache handles the cache-memory interface
- Measuring cache performance

$$\begin{aligned} AMAT &= HitTime_{L1} + MissRate_{L1} \times MissPenalty_{L1} \\ &= HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}) \end{aligned}$$

- Local miss rate
  - misses in this cache divided by the total number of memory accesses to this cache ($MissRate_{L2}$)
- Global miss rate (& biggest penalty!)
  - misses in this cache divided by the total number of memory accesses generated by the CPU ($MissRate_{L1} \times MissRate_{L2}$)

Local vs Global Misses

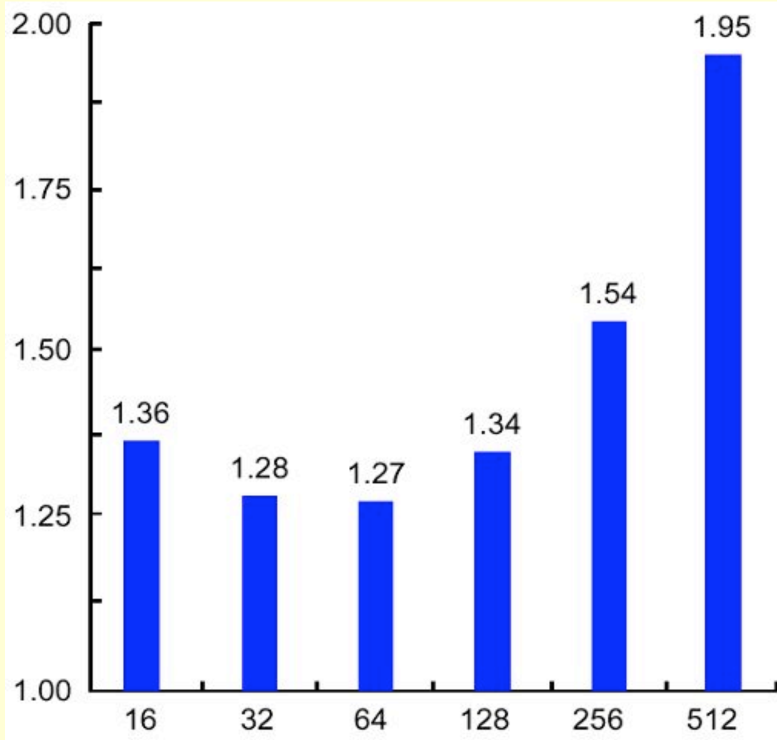


(Global miss rate close to single level cache rate provided L2 >> L1)

L2 Cache Parameters

- 32 bit bus
- 512KB cache

Relative execution time



Block size of second-level cache (byte)

- L1 cache directly affects the processor design and clock cycle: should be simple and small
- Bulk of optimization techniques can go easily to L2 cache
- Miss-rate reduction more practical for L2
- Considering the L2 cache can improve the L1 cache design,
  - e.g. use write-through if L2 cache applies write-back

Reducing Hit Time

Average Access Time = Hit Time x (1 - Miss Rate) + Miss Time x Miss Rate

- Hit rate is typically very high compared to miss rate
  - any reduction in hit time is magnified
- Hit time critical: affects processor clock rate
- Three techniques to reduce hit time:
  - Simple and small caches
  - Avoid address translation during cache indexing
  - Pipelining writes for fast write hits

<u>Simple and small caches</u>

- Design simplicity limits control logic complexity and allows shorter clock cycles
- On-chip integration decreases signal propagation delay, thus reducing hit time
  - Alpha 21164 has 8KB Instruction and 8KB data cache and 96KB second level cache to reduce clock rate

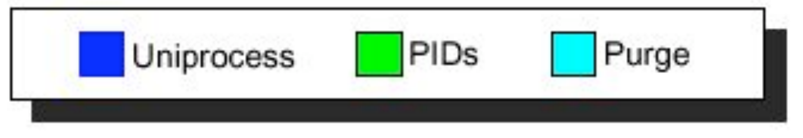
Avoiding Address Translation

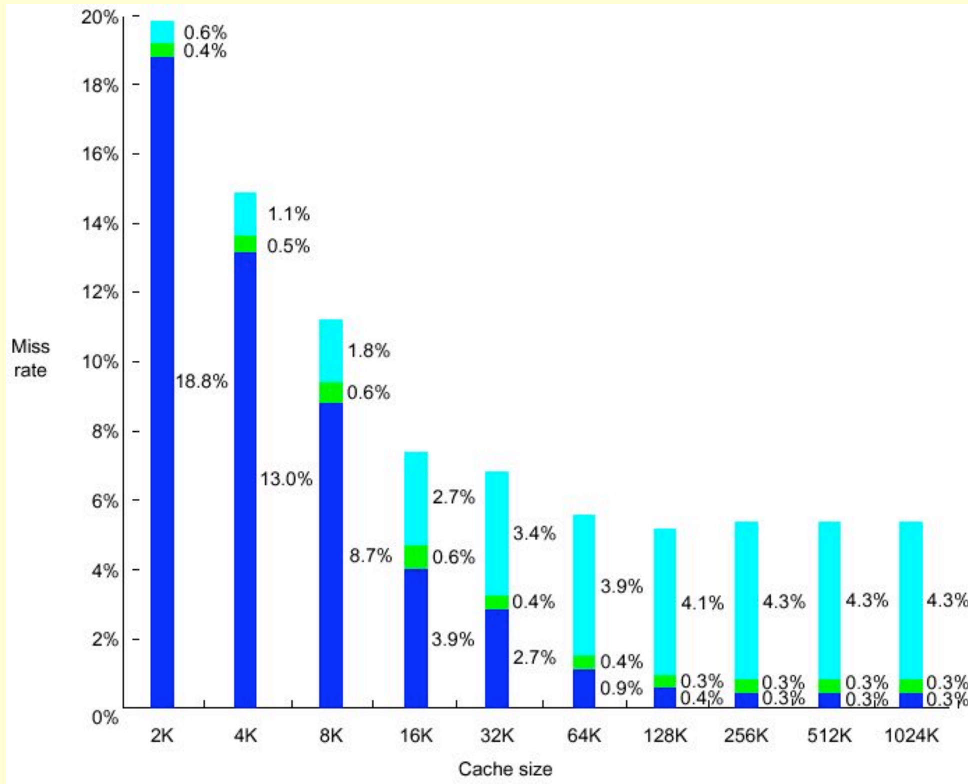
- Send virtual address to cache?
  - Called Virtually Addressed Cache or just Virtual Cache vs. Physical Cache
  - Every time process is switched logically must flush the cache; otherwise get false hits
    - Cost is time to flush + "compulsory" misses from empty cache
  - Dealing with aliases (sometimes called synonyms)
    - Two different virtual addresses map to same physical address causing unnecessary read misses or even RAW
  - I/O must interact with cache, so need virtual address

Solutions

- Solution to aliases
  - HW guarantees that every cache block has unique physical address (simply check all cache entries)
  - SW guarantee: lower n bits must have same address so that it overlaps with index; as long as covers index field & direct mapped, they must be unique; called page coloring
- Solution to cache flush
  - Add process identifier tag that identifies process as well as address within process: cannot get a hit if wrong process

Impact of Using Process ID

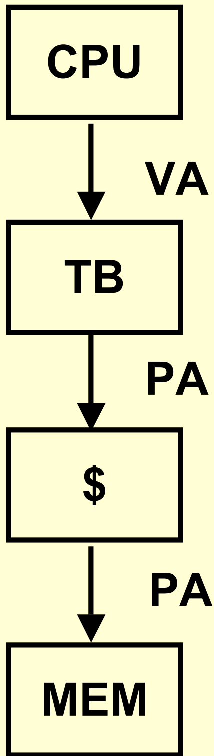




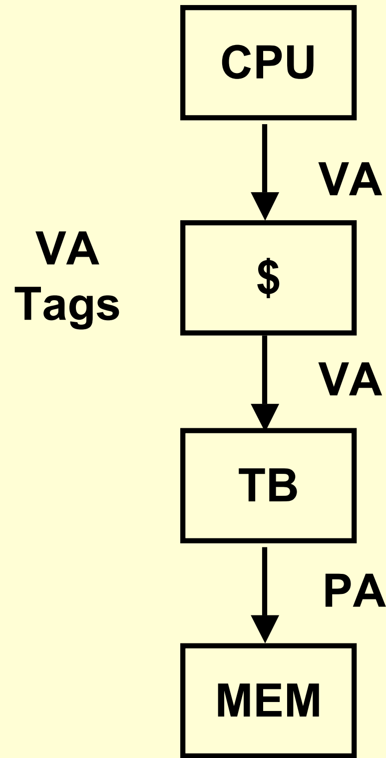
- Miss rate vs. virtually addressed cache size of a program measured three ways:
  - Without process switches (uniprocessor)
  - With process switches using a PID tag (PID)
  - With process switches but without PID (purge)

Virtually Addressed Caches

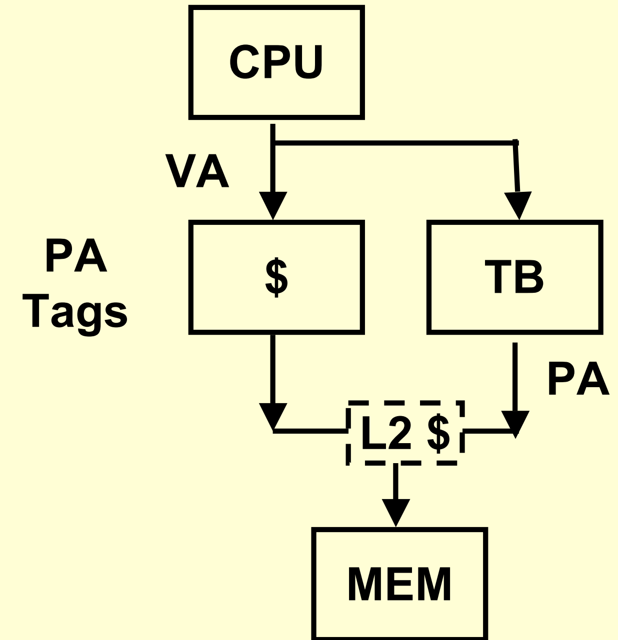
VA: Virtual address **TB**: Translation buffer **PA**: Page address



Conventional Organization



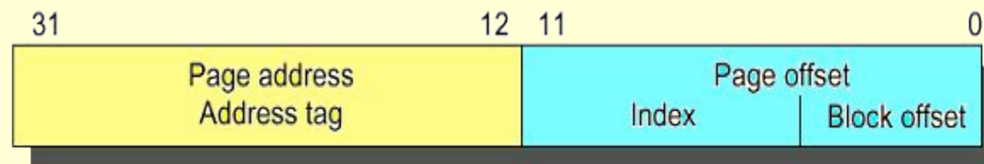
Virtually Addressed Cache
Translate only on miss
Synonym Problem



Overlap $ access with VA translation: requires $ index to remain invariant across translation

Indexing via Physical Addresses

- If index is physical part of address, can start tag access in parallel with translation
- To get the best of the physical and virtual caches, use the page offset (not affected by the address translation) to index the cache
- The drawback is that direct-mapped caches cannot be bigger than the page size (typically 4-KB)

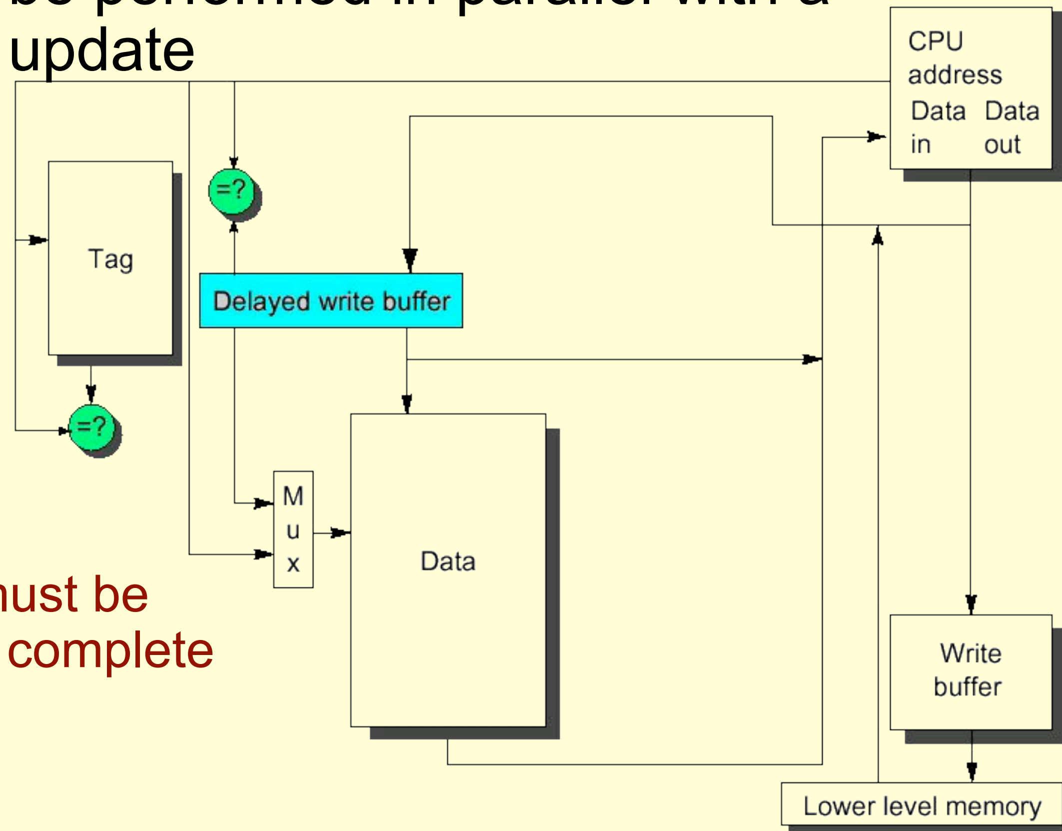


- To support bigger caches and use same technique:
  - Use higher associativity since the tag size gets smaller
  - OS implements page coloring since it will fix a few least significant bits in the address (move part of the index to the tag)

Pipelined Cache Writes

- In cache read, tag check and block reading are performed in parallel while writing requires validating the tag first
- Tag Check can be performed in parallel with a previous cache update

Pipeline Tag Check and Update Cache as separate stages; current write tag check & previous write cache update



"Delayed Write Buffer"; must be checked on reads; either complete write or read from buffer

Cache Optimization Summary

	Technique	***MR***	***MP***	***HT***	***Complexity***
miss rate	Larger Block Size	+	–		0
	Higher Associativity	+		–	1
	Victim Caches	+			2
	Pseudo-Associative Caches	+			2
	HW Pre-fetching of Instr/Data	+			2
	Compiler Controlled Pre-fetching	+			3
	Compiler Reduce Misses	+			0
miss penalty	Priority to Read Misses		+		1
	Sub-block Placement		+	+	1
	Early Restart & Critical Word 1st		+		2
	Non-Blocking Caches		+		3
	Second Level Caches		+		2
hit time	Small & Simple Caches	–		+	0
	Avoiding Address Translation			+	2
	Pipelining Writes			+	1

Memory Hierarchy

Capacity
Access Time

Upper Level

Staging
Transfer Unit

faster

CPU Registers
100s Bytes
<10s ns

Registers

Instr. Operands

Prog./compiler
1-8 bytes

Cache
K Bytes
10-40 ns

Cache

Blocks

cache cntl
8-128 bytes

Main Memory
M Bytes
70ns-1us

Main Memory

Pages

OS
512-4K bytes

Disk
G Bytes
ms

Disk

Files

user/operator
Mbytes

Larger

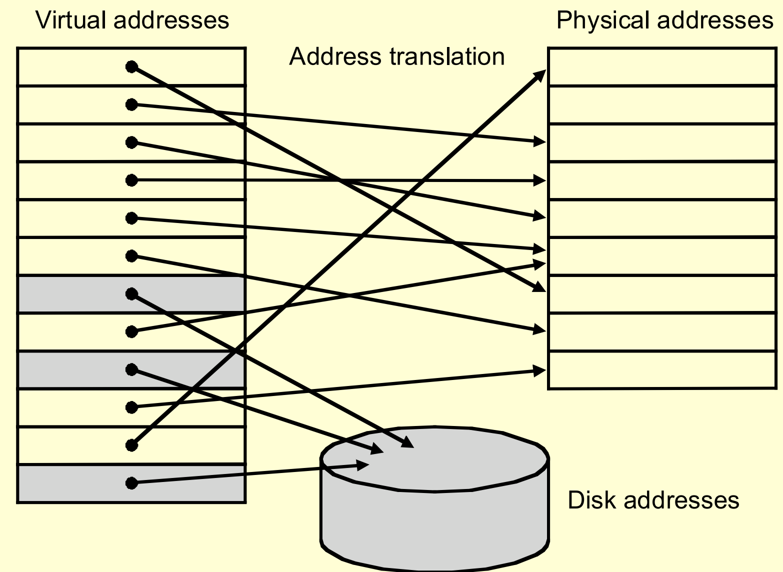
Tape
infinite
sec-min

Tape

Lower Level

Virtual Memory

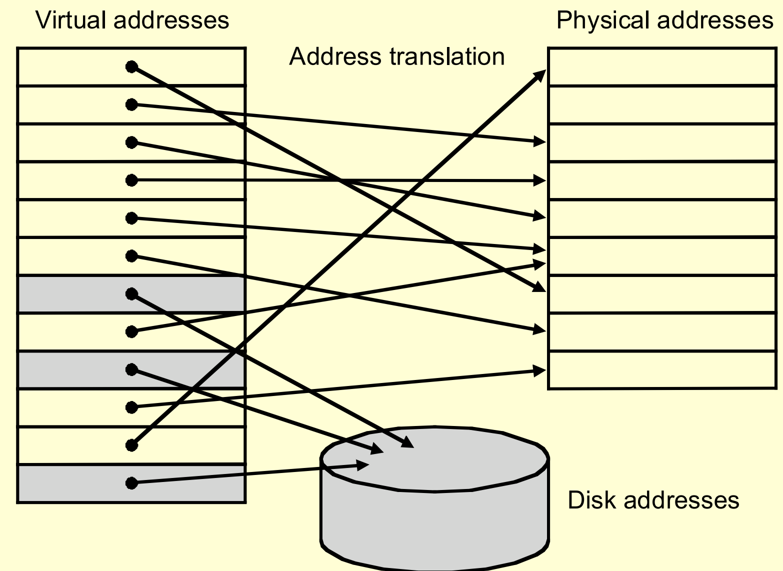
- Using virtual addressing, main memory plays the role of cache for disks
- The virtual space is much larger than the physical memory space
- Physical main memory contains only the active portion of the virtual space
- Address space can be divided into fixed size (pages) or variable size (segments) blocks



Cache		***Virtual memory***
Block	⇒	Page
Cache miss	⇒	page fault
Block addressing	⇒	Address translation

Virtual Memory

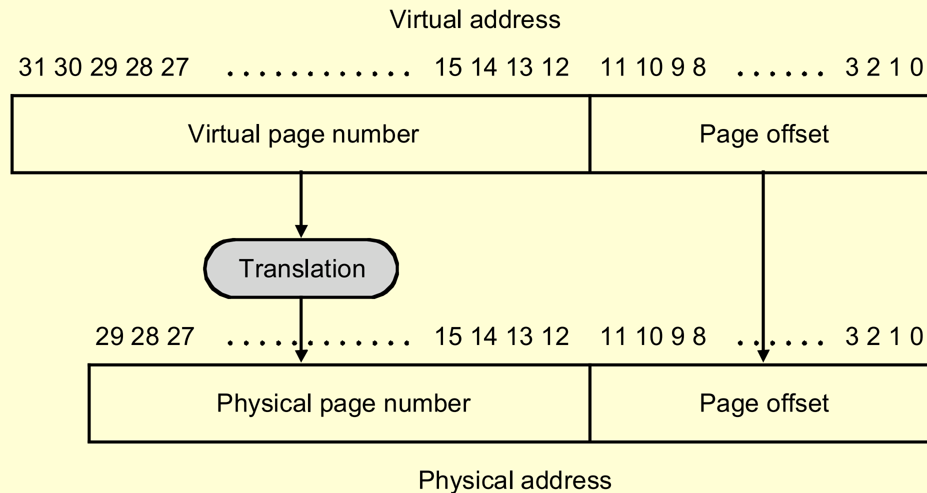
- Advantages
  - Allows efficient and safe data sharing of memory among multiple programs
  - Moves programming burdens of a small, limited amount of main memory
  - Simplifies program loading and avoid the need for contiguous memory block
  - allows programs to be loaded at any physical memory location



Cache		***Virtual memory***
Block	⇒	Page
Cache miss	⇒	page fault
Block addressing	⇒	Address translation

Virtual Addressing

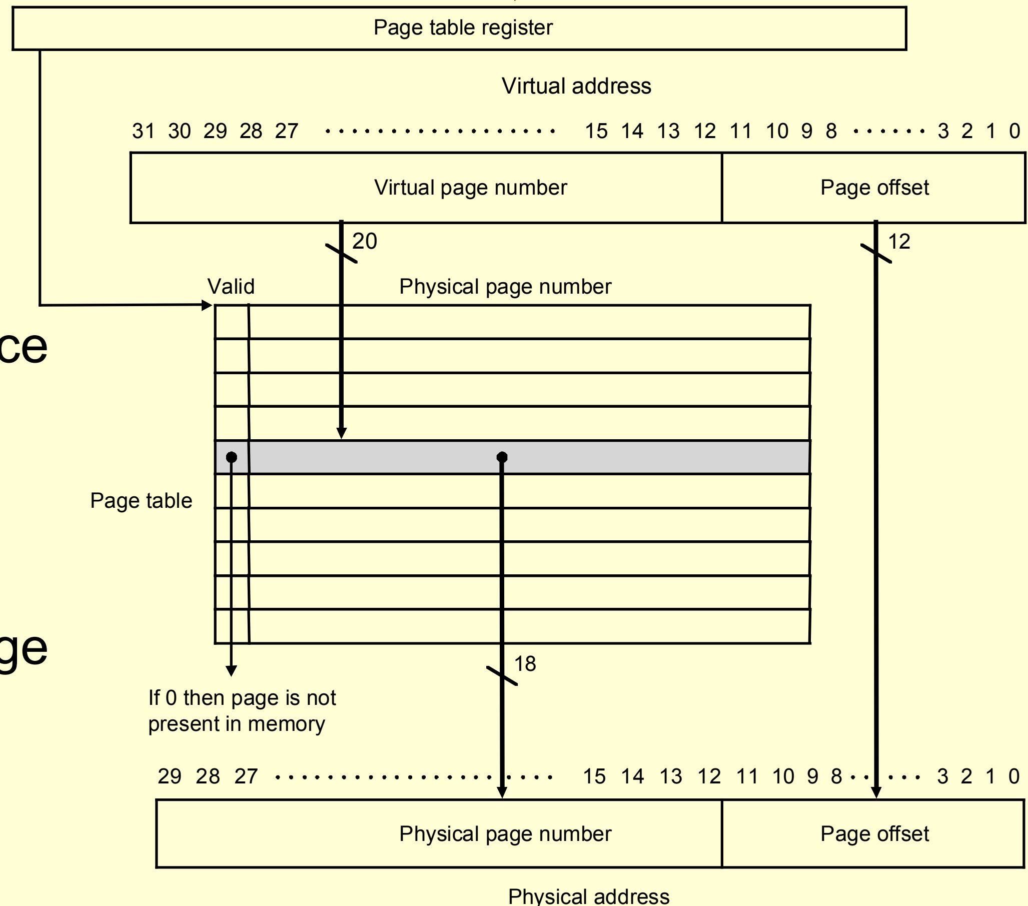
- Page faults are costly and take millions of cycles to process (disks are slow)
- Optimization Strategies:
  – Pages should be large enough to amortize the access time
  – Fully associative placement of pages reduces page fault rate
  – Software-based so can use clever page placement
  – Write-through can make writing very time consuming (use copy back)



Page Table

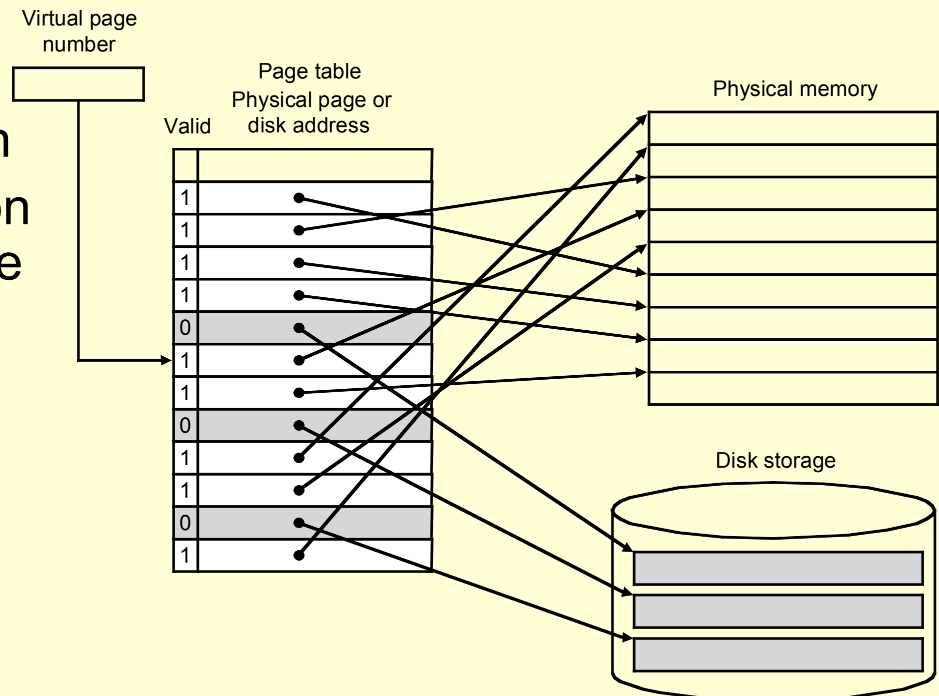
Hardware supported

- Page table:
  - Resides in main memory
  - One entry per virtual page
  - No tag is requires since it covers all virtual pages
  - Point directly to physical page
  - Table can be very large
  - Operating sys. may maintain one page table per process
  - A dirty bit is used to track modified pages for copy back



Page Faults

- A page fault happens when the valid bit of a virtual page is off
- A page fault generates an exception to be handled by the operating system to bring the page to main memory from a disk
- The operating system creates space for all pages on disk and keeps track of the location of pages in main memory and disk
- Page location on disk can be stored in page table or in an auxiliary structure
- LRU page replacement strategy is the most common
- Simplest LRU implementation uses a reference bit per page and periodically reset reference bits



Optimizing Page Table Size

With a 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry:

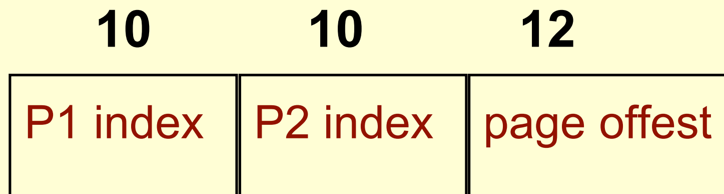
$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \text{ page table entries} \times 2^{2} \frac{\text{bytes}}{\text{page table entry}} = 4\,\text{MB}$$

- Optimization techniques:
  - Keep bound registers to limit the size of page table for given process in order to avoid empty slots
  - Store only physical pages and apply hashing function of the virtual address (inverted page table)
  - Use multi-level page table to limit size of the table residing in main memory
  - Allow paging of the page table
  - Cache the most used pages $\Rightarrow$ Translation Look-aside Buffer

Multi-Level Page Table

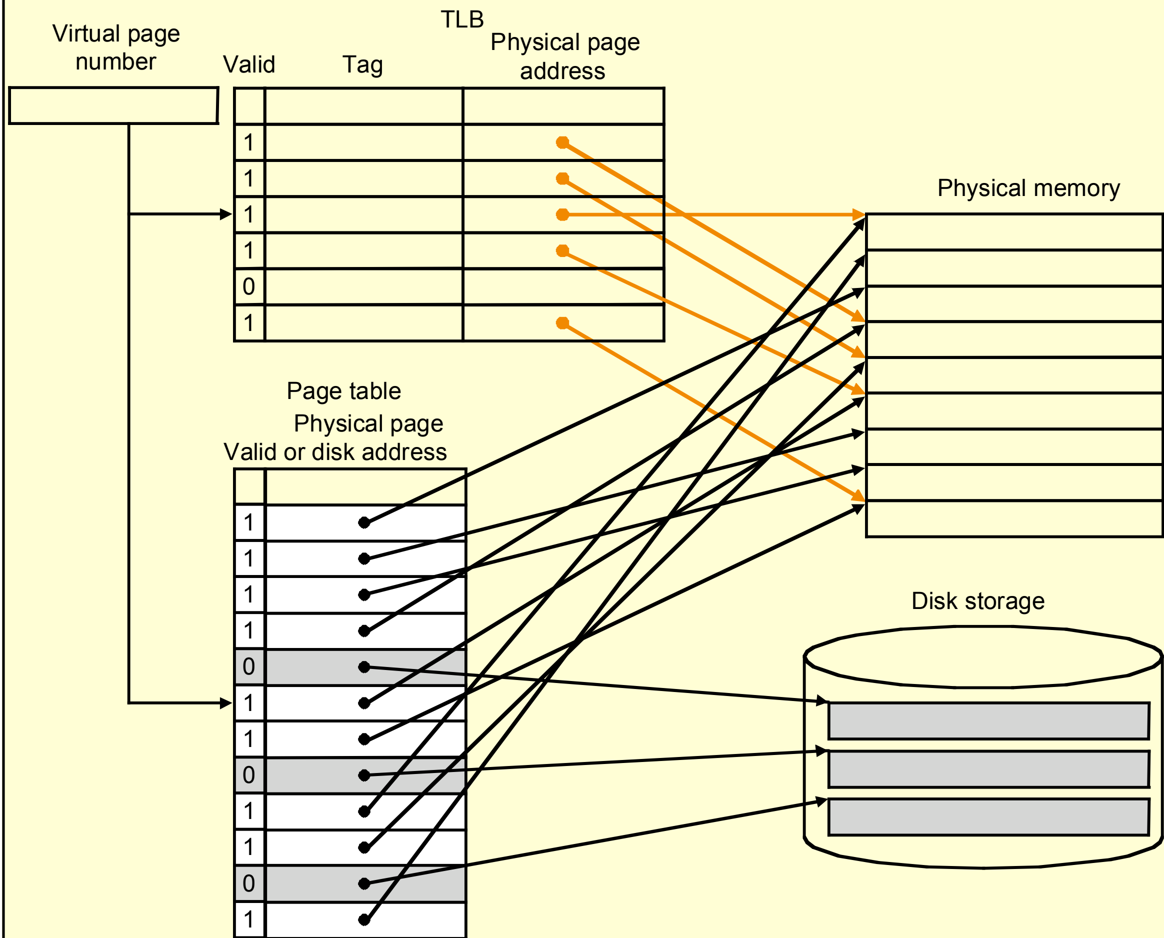
32-bit address:



- 2 GB virtual address space
- 4 MB of PTE2
  - paged, holes
- 4 KB of PTE1

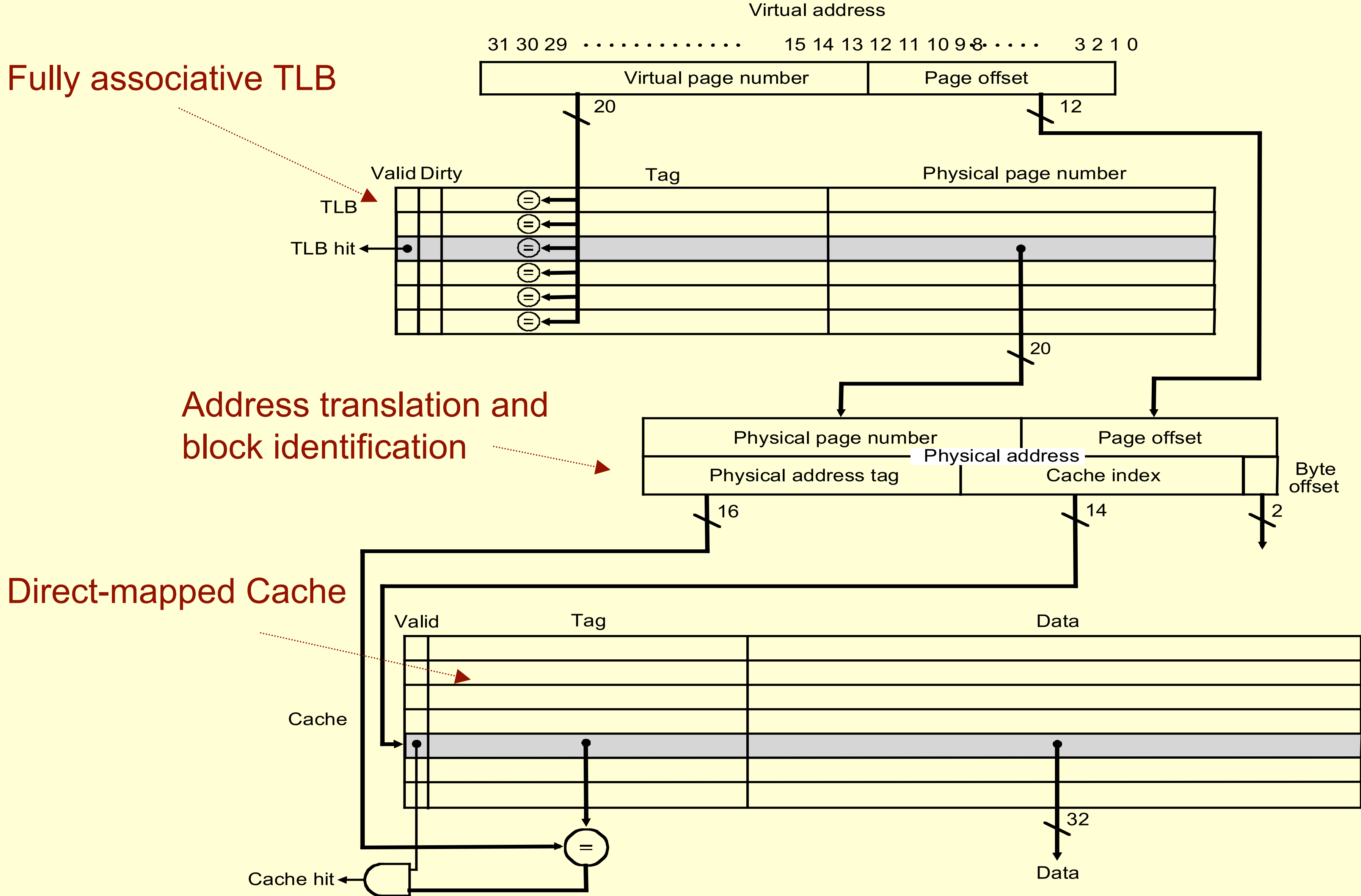
Inverted page table can be the only practical solution for huge address space, e.g 64-bit address space

Translation Look-aside Buffer

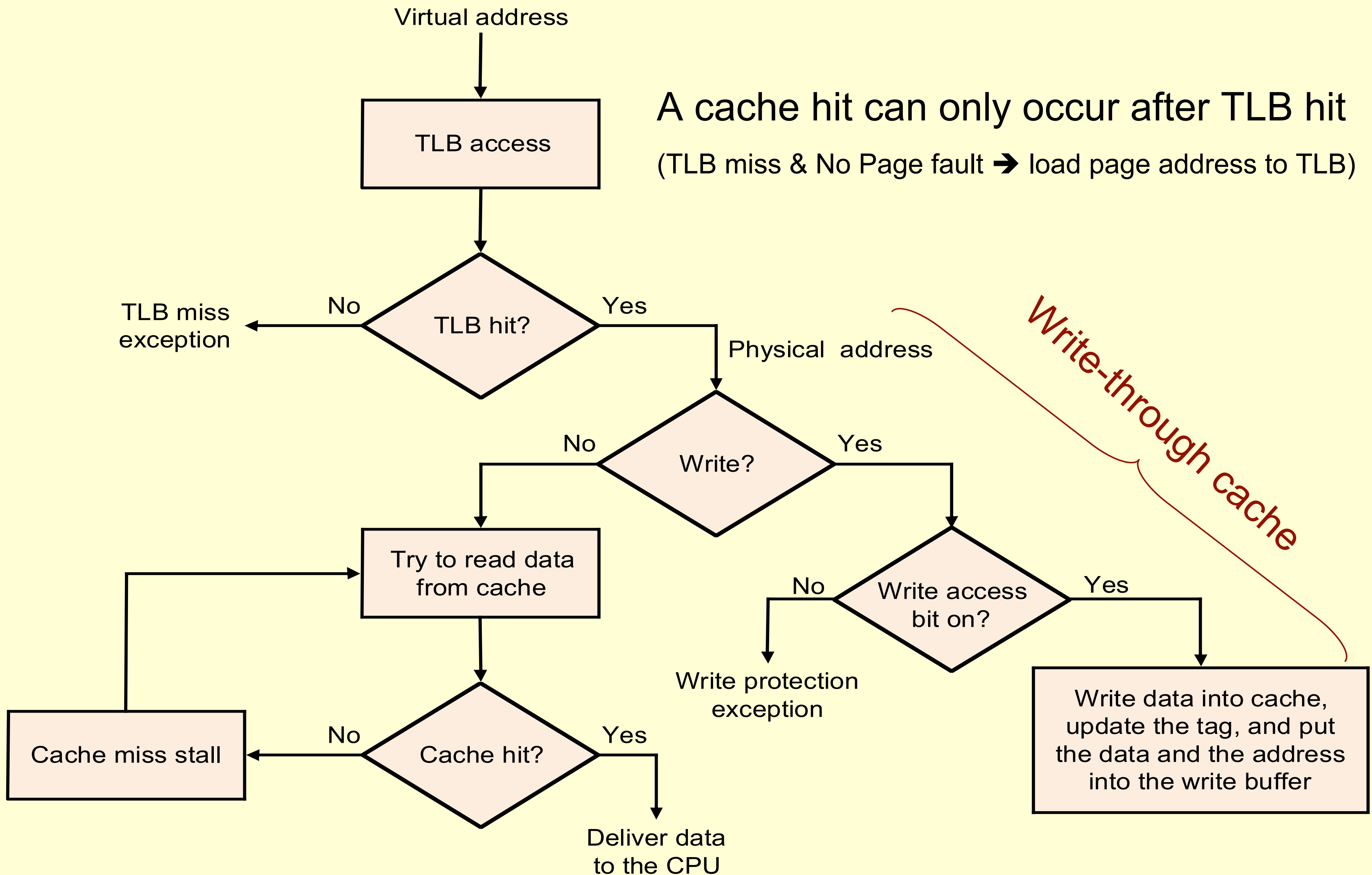


- Special cache for recently used translation
- TLB misses are typically handled as exceptions by operating system
- Simple replacement strategy since TLB misses happen frequently

TLB and Cache in MIPS



TLB and Cache in MIPS



Memory Related Exceptions

Possible exceptions:

Cache miss: referenced block not in cache and needs to be fetched from main memory

TLB miss: referenced page of virtual address needs to be checked in the page table

Page fault: referenced page is not in main memory and needs to be copied from disk

Cache	TLB	Page fault	Possible? If so, under what condition
miss	hit	hit	Possible, although the page table is never really checked if TLB hits
hit	miss	hit	TLB misses, but entry found in page table and data found in cache
miss	miss	hit	TLB misses, but entry found in page table and data misses in cache
miss	miss	miss	TLB misses and followed by page fault. Data must miss in cache
miss	hit	miss	Impossible: cannot have a translation in TLB if page is not in memory
hit	hit	miss	Impossible: cannot have a translation in TLB if page is not in memory
hit	miss	miss	Impossible: data is not allowed in cache if page is not in memory

Memory Protection

- Want to prevent a process from corrupting memory space of other processes
  - Privileged and non-privileged execution
- Implementation can map independent virtual pages to separate physical pages
- Write protection bits in the page table for authentication
- Sharing pages through mapping virtual pages of different processes to same physical pages

Memory Protection

- To enable the operating system to implement protection, the hardware must provide at least the following capabilities:
  - Support at least two mode of operations, one of them is a user mode
  - Provide a portion of CPU state that a user process can read but not write,
    - e.g. page pointer and TLB
  - Enable change of operation modes through special instructions

Handling TLB Misses & Page Faults

- TLB Miss: (hardware-based handling)
  - Check if the page is in memory (valid bit) → update the TLB
  - Generate page fault exception if page is not in memory
- Page Fault: (handled by operating system)
  - Transfer control to the operating system
  - Save processor status: registers, program counter, page table pointer, etc.
  - Lookup the page table and find the location of the page on disk
  - Choose a physical page to host the referenced page, if the candidate physical page is modified (dirty bit is set) the page needs to be written back
  - Start reading the page from disk to the assigned physical page
- The processor needs to support "restarting" instructions in order to guarantee correct execution
  - The user process causing the page fault will be suspended by the operating system until the page is readily available in main memory
  - Protection violations are handled by the operating system similarly but without automatic instruction restarting