



**COMP9242**  
**Advanced Operating Systems**  
**S2/2011 Week 2:**  
**Caches:**  
**What every OS Designer Must Know**



**Australian Government**  
**Department of Broadband, Communications**  
**and the Digital Economy**  
**Australian Research Council**

**NICTA Funding and Supporting Members and Partners**



# Copyright Notice

---

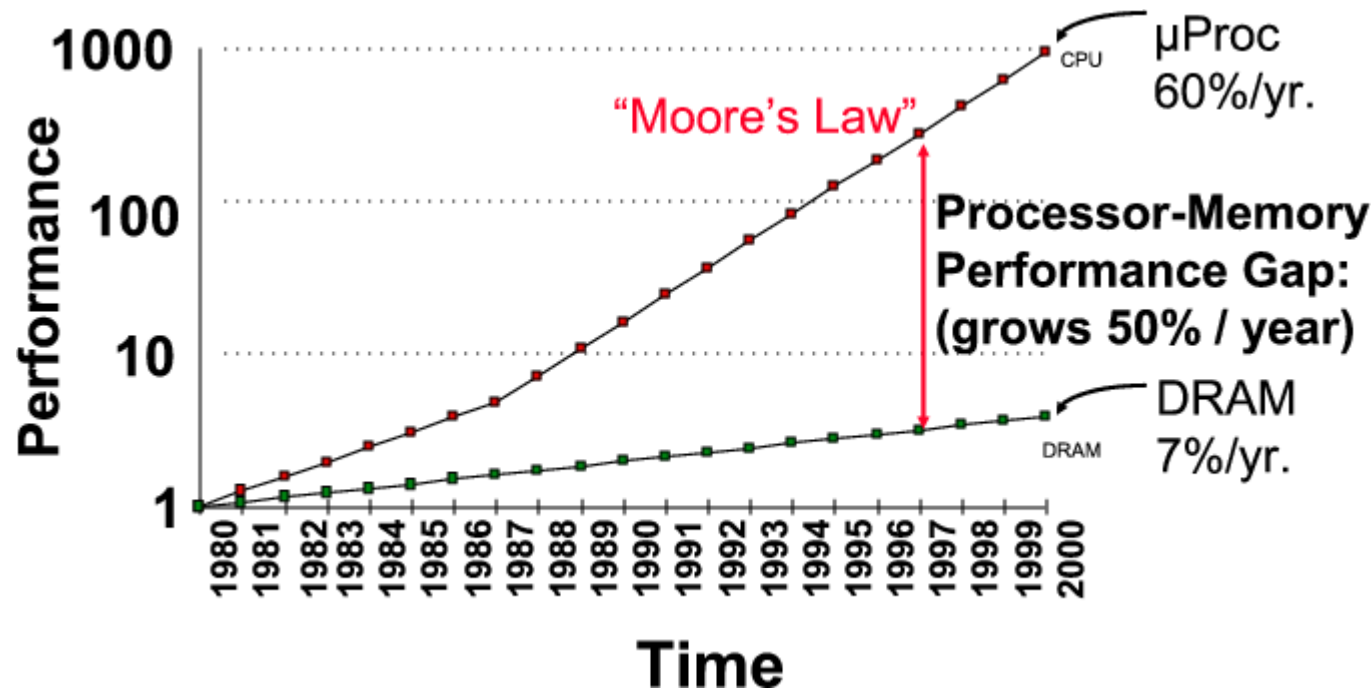


**These slides are distributed under the Creative Commons Attribution 3.0 License**

- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work
- under the following conditions:
  - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    - “Courtesy of Gernot Heiser, [Institution]”, where [Institution] is one of “UNSW” or “NICTA”

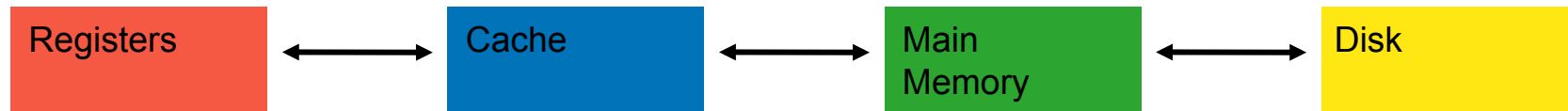
The complete license text can be found at  
<http://creativecommons.org/licenses/by/3.0/legalcode>

# The Memory Wall



Multicore offsets stagnant per-core performance with proliferation of cores  
→ Basic trend is unchanged

# Caching



- Cache is fast (1–5 cycle access time) memory sitting between fast registers and slow RAM (10–100 cycles access time)
- Holds recently-used data or instructions to save memory accesses
- Matches slow RAM access time to CPU speed if high **hit rate** (> 90%)
- Is hardware maintained and (mostly) transparent to software
- Sizes range from few KiB to several MiB.
- Usually a hierarchy of caches (2–5 levels), on- and off-chip

Good overview of implications of caches for operating systems: [Schimmel 94]

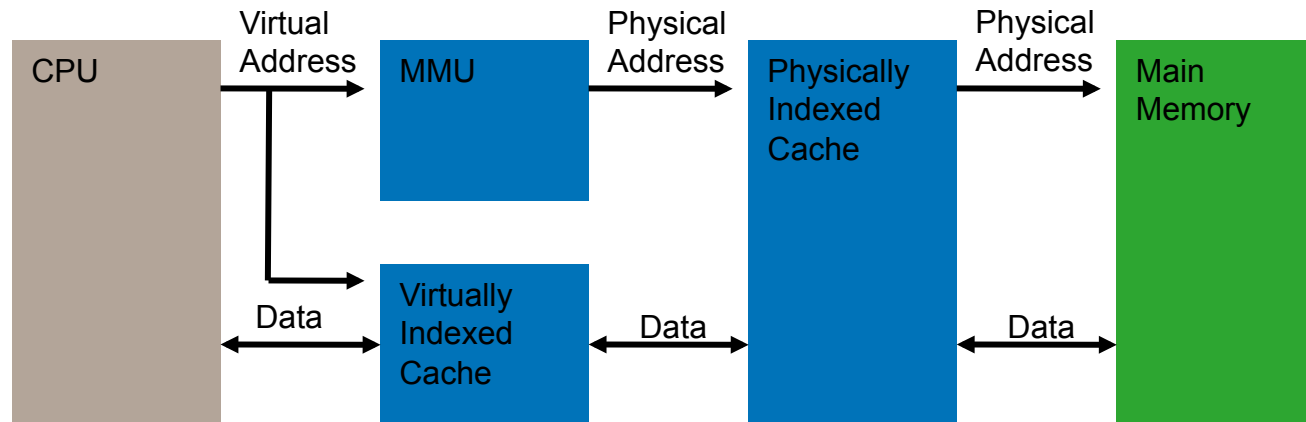
# Cache Organization

---



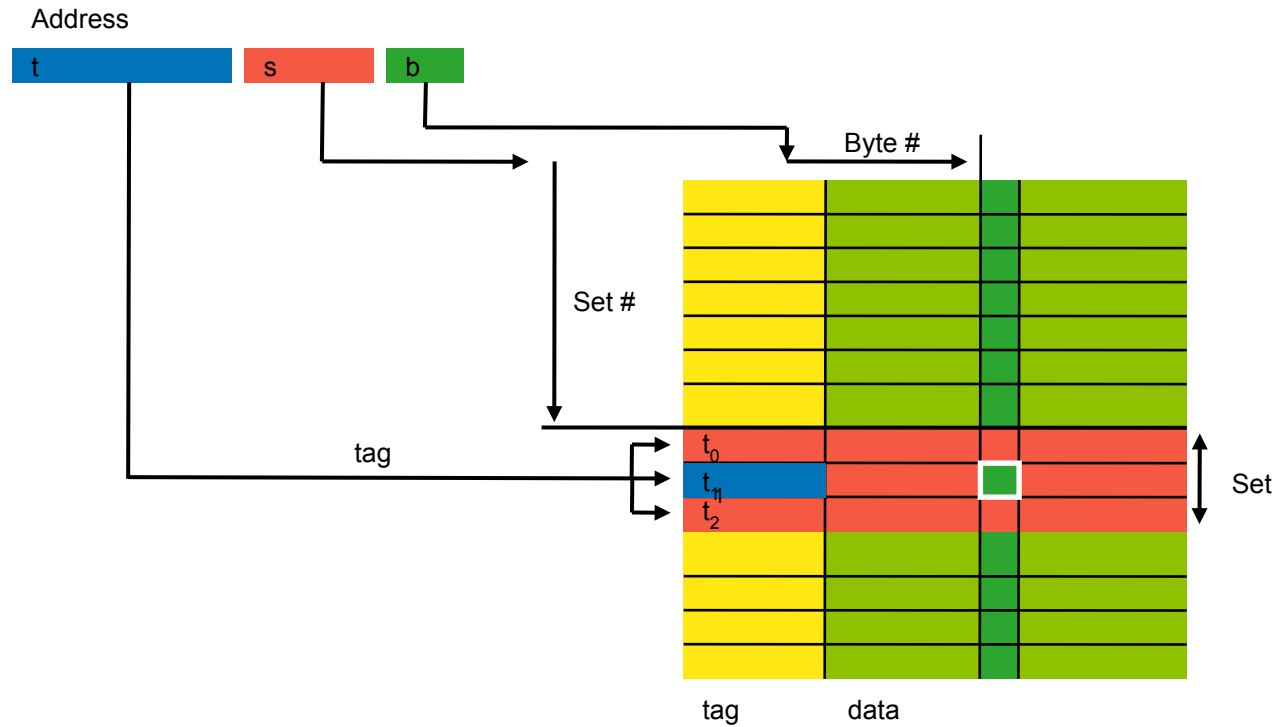
- Data transfer unit between registers and L1 cache:  $\leq 1$  word (1–16B)
- Cache *line* is transfer unit between cache and RAM (or slower cache)
  - typically 16–32 bytes, sometimes 128 bytes and more
- Line is also unit of storage allocation in cache
- Each line has associated control info:
  - valid bit
  - modified bit
  - tag
- Cache improves memory access by:
  - absorbing most reads (increases bandwidth, reduces latency)
  - making writes asynchronous (hides latency)
  - clustering reads and writes (hides latency)

# Cache Access



- **Virtually indexed:** looked up by *virtual address*
  - operates concurrently with address translation
- **Physically indexed:** looked up by *physical address*
  - requires result of address translation

# Cache Indexing



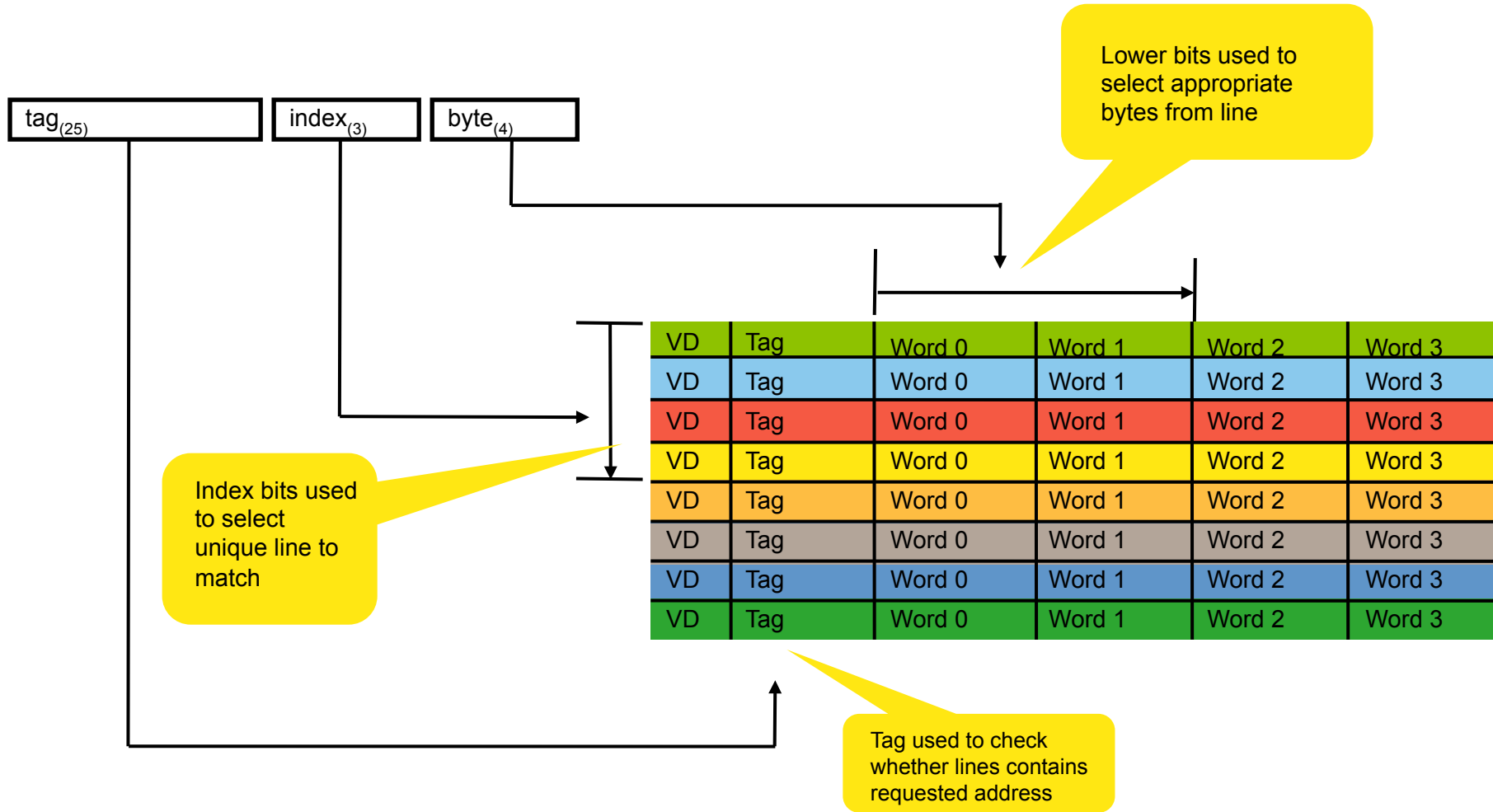
- The *tag* is used to distinguish lines of set...
- Consists of high-order bits not used for indexing

# Cache Indexing

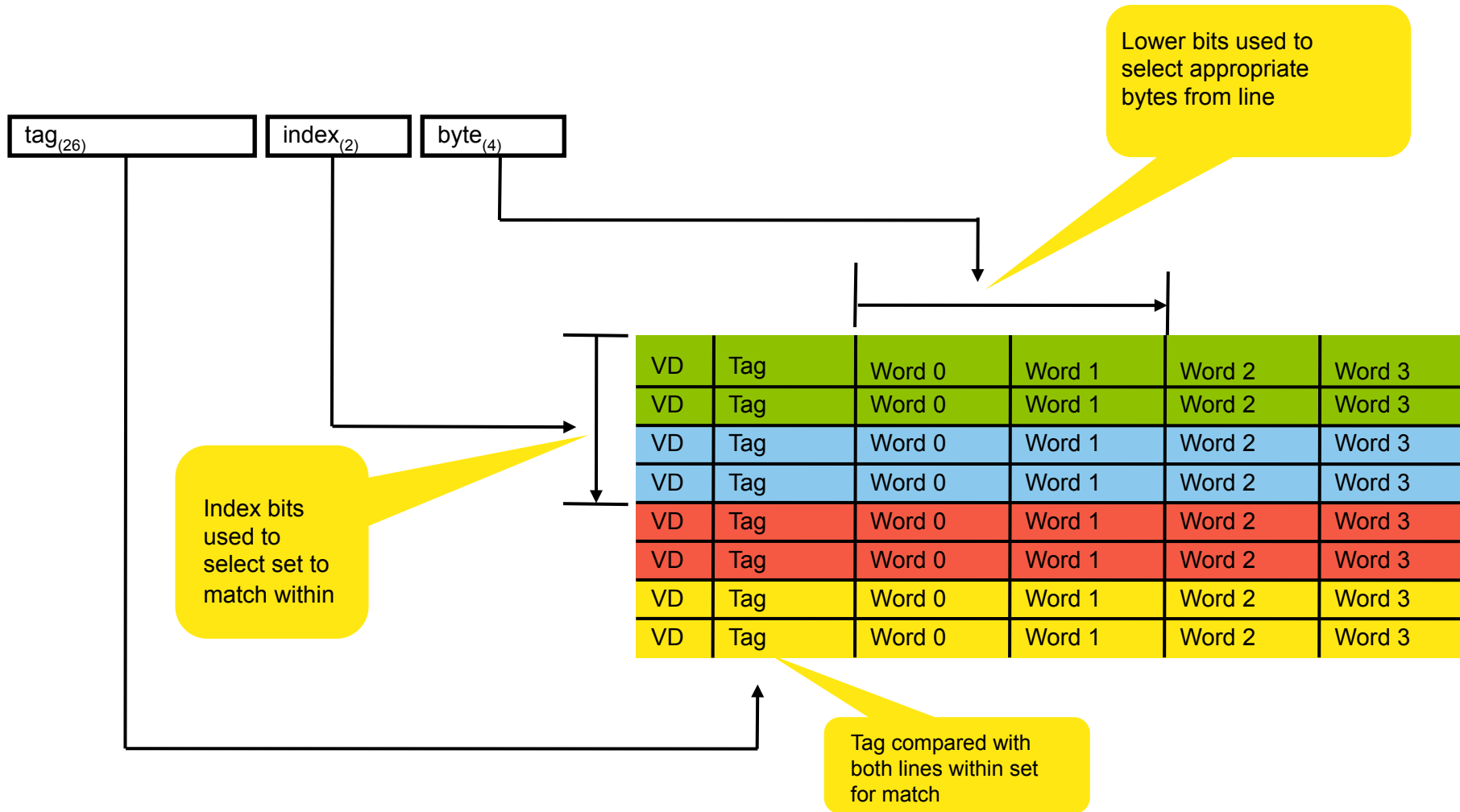


- Address is hashed to produce index of *line set*.
- Associative lookup of line within set
- $n$  lines per set:  *$n$ -way set-associative cache*.
  - typically  $n = 1 \dots 5$ , some embedded processors use 32–64
  - $n = 1$  is called *direct mapped*.
  - $n = \infty$  is called *fully associative* (unusual for CPU caches)
- Hashing must be simple (complex hardware is slow)
  - use least-significant bits of address

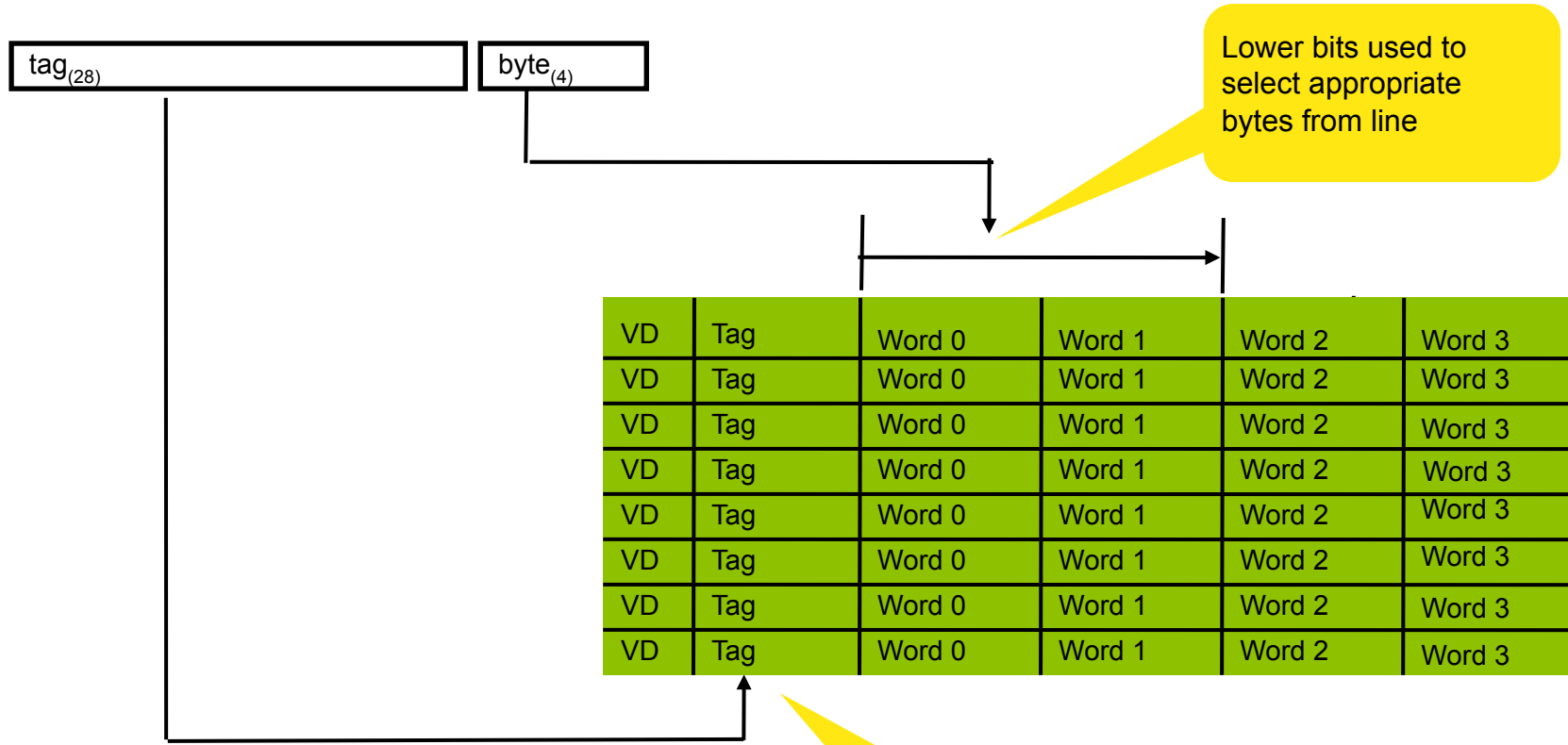
# Cache Indexing: Direct Mapped



# Cache Indexing: 2-Way Associative



# Caching Index: Fully Associative



Lower bits used to select appropriate bytes from line

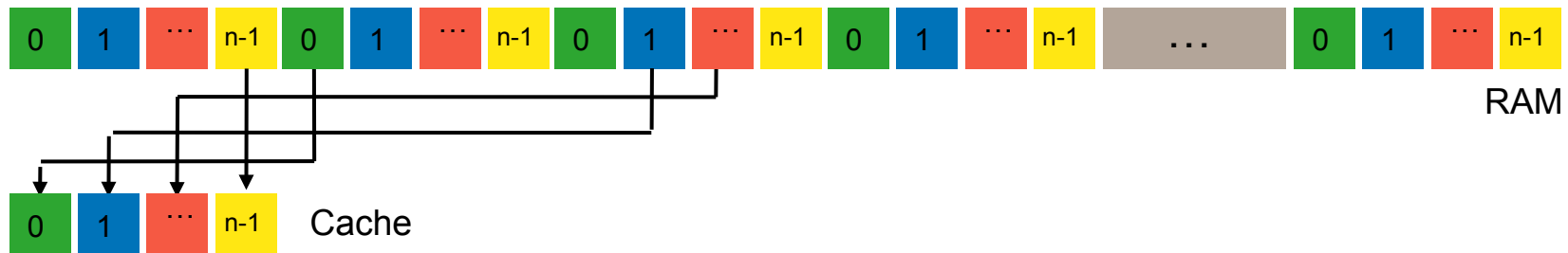
Tag compared with all lines for a match

Note: Lookup hardware for many tags is large and slow ⇒ does not scale

# Cache Mapping



→ Different memory locations map to same cache line:



→ Locations mapping to cache set #  $i$  are said to be of *colour  $i$*

→  *$n$ -way* associative cache can hold  $n$  lines of the same colour

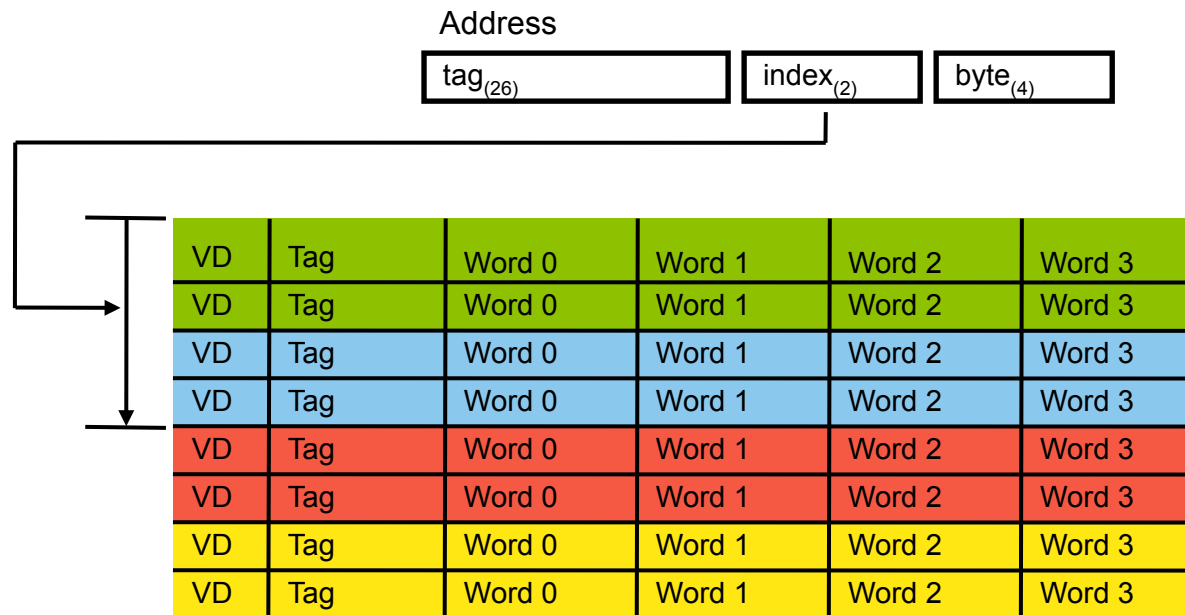
→ Types of cache misses:

- **Compulsory miss:** data cannot be in cache (of infinite size)
  - first access (after flush)
- **Capacity miss:** all cache entries are in use by other data
- **Conflict miss:** set of the right colour is full
  - miss that would not happen on fully-associative cache
- **Coherence miss:** miss forced by hardware coherence protocol
  - multiprocessors

# Cache Replacement Policy



- Indexing (using address) points to specific line set.
- On miss: all lines of set valid ⇒ must *replace existing line*.
- Replacement strategy must be simple (hardware)
  - Dirty bit determines whether line needs to be written back
  - Typical policies:
    - pseudo-LRU
    - FIFO
    - random
    - toss clean



# Cache Write Policy

---



## → Treatment of store operations:

- **write back:** Stores update cache only  
memory is updated once dirty line is replaced (flushed)
  - ✓ clusters writes
  - ✗ memory is inconsistent with cache
  - ✗ unsuitable for (most) multiprocessor designs
- **write through:** Stores update cache and memory immediately
  - ✓ memory is always consistent with cache
  - ✗ increased memory/bus traffic

## → On store to a line not presently in cache, use:

- **write allocate:** allocate a cache line to the data and store  
-typically requires reading line into cache first!
- **no allocate:** store to memory and bypass cache

## → Typical combinations:

- write-back & write-allocate
- write-through & no-allocate

# Cache Addressing Schemes

---

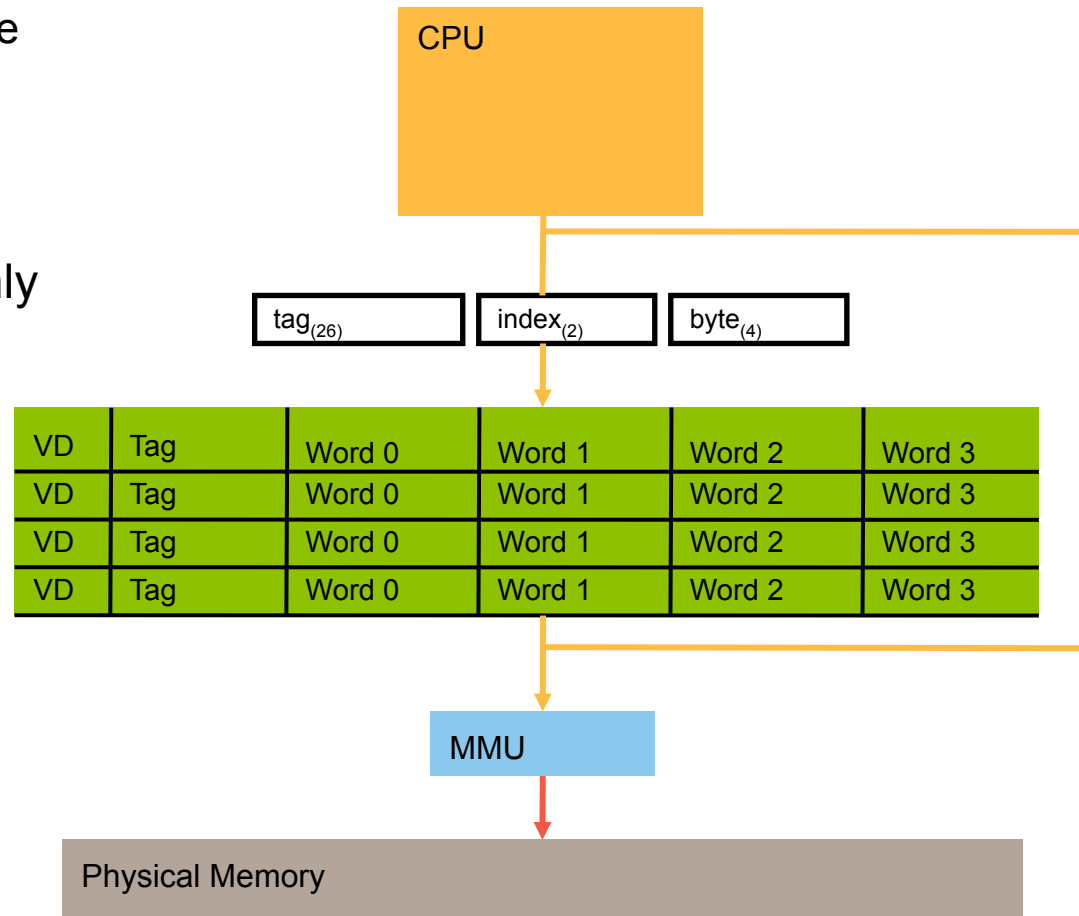


- For simplicity, discussion so far assumed cache sees only one kind of address: virtual or physical
- However, *indexing and tagging can use different addresses*
- Four possible addressing schemes:
  - virtually-indexed, virtually-tagged (VV) cache
  - virtually-indexed, physically-tagged (VP) cache
  - physically-indexed, virtually-tagged (PV) cache
  - physically-indexed, physically-tagged (PP) cache
- PV caches can only make sense with complex and unusual MMU designs
  - not considered here any further

# Virtually-Indexed, Virtually-Tagged Cache



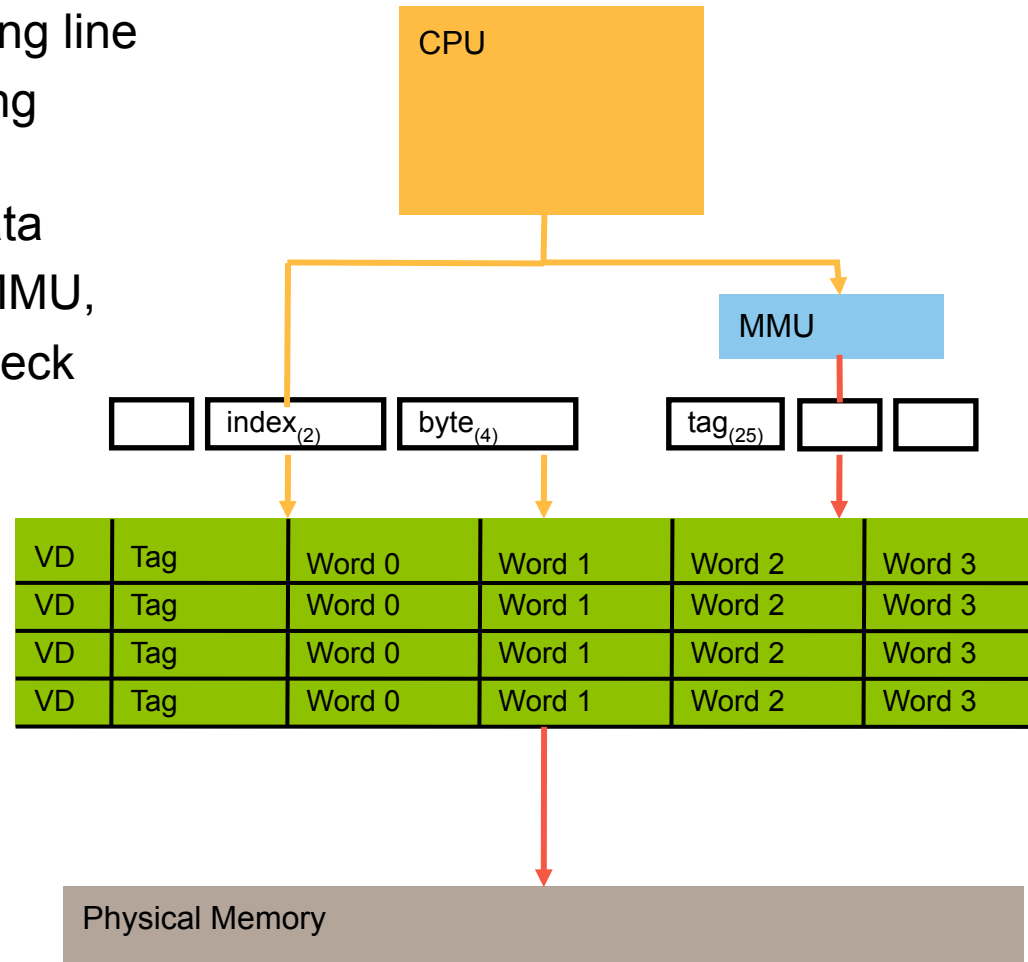
- Also called
  - virtually-addressed cache
- Also (incorrectly) called
  - virtual cache
  - virtual address cache
- Uses virtual addresses only
  - can operate concurrently with MMU
  - still needs MMU lookup to determine access rights
- Used for on-core L1



# Virtually-Indexed, Physically-Tagged Cache



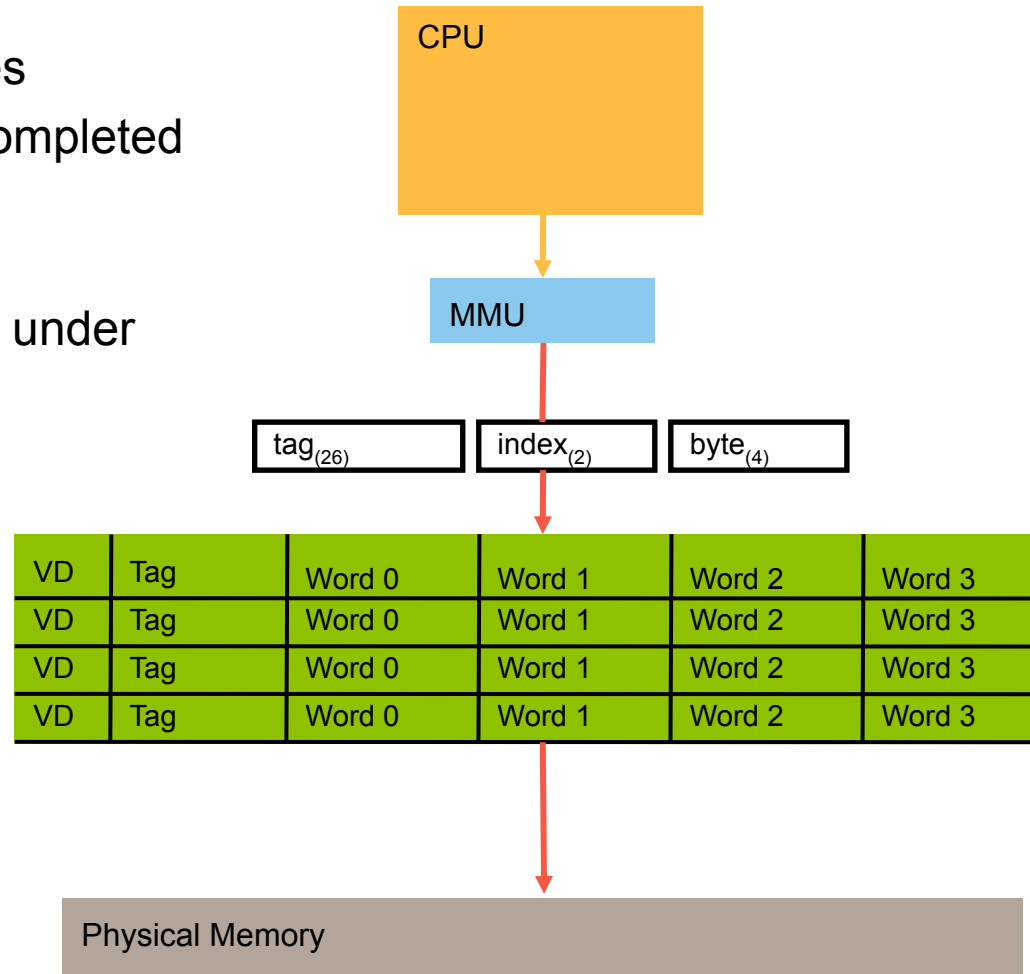
- Virtual address for accessing line
- Physical address for tagging
- Needs address translation completed for retrieving data
- Indexing concurrent with MMU, use MMU output for tag check
- Typically used for on-core L1



# Physically-Indexed, Physically-Tagged Cache



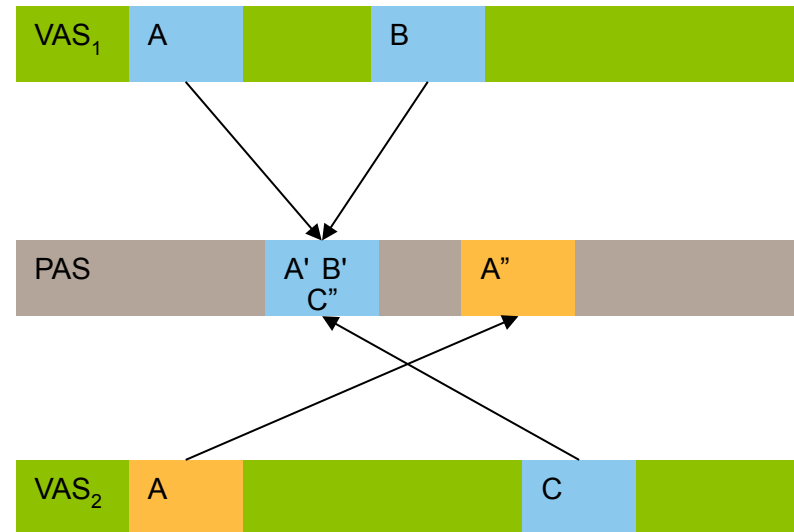
- Only uses physical addresses
- Needs address translation completed before begin of access
- Typically used off-core
- Note: page offset is invariant under virtual-address translation
  - if index bits are *subset* of offset, PP cache can be accessed without result of translation!
  - VP and PP cache become the same in this case
  - fast and suitable for on-core use (esp. L1)



# Cache Issues



- Caches are managed by hardware transparent to software
  - OS doesn't have to worry about them, ~~right?~~ Wrong!
- Software-visible cache effects:
  - performance
  - *homonyms*:
    - same name, different data
    - can affect correctness!
  - *synonyms*:
    - different name, same data
    - can affect correctness!

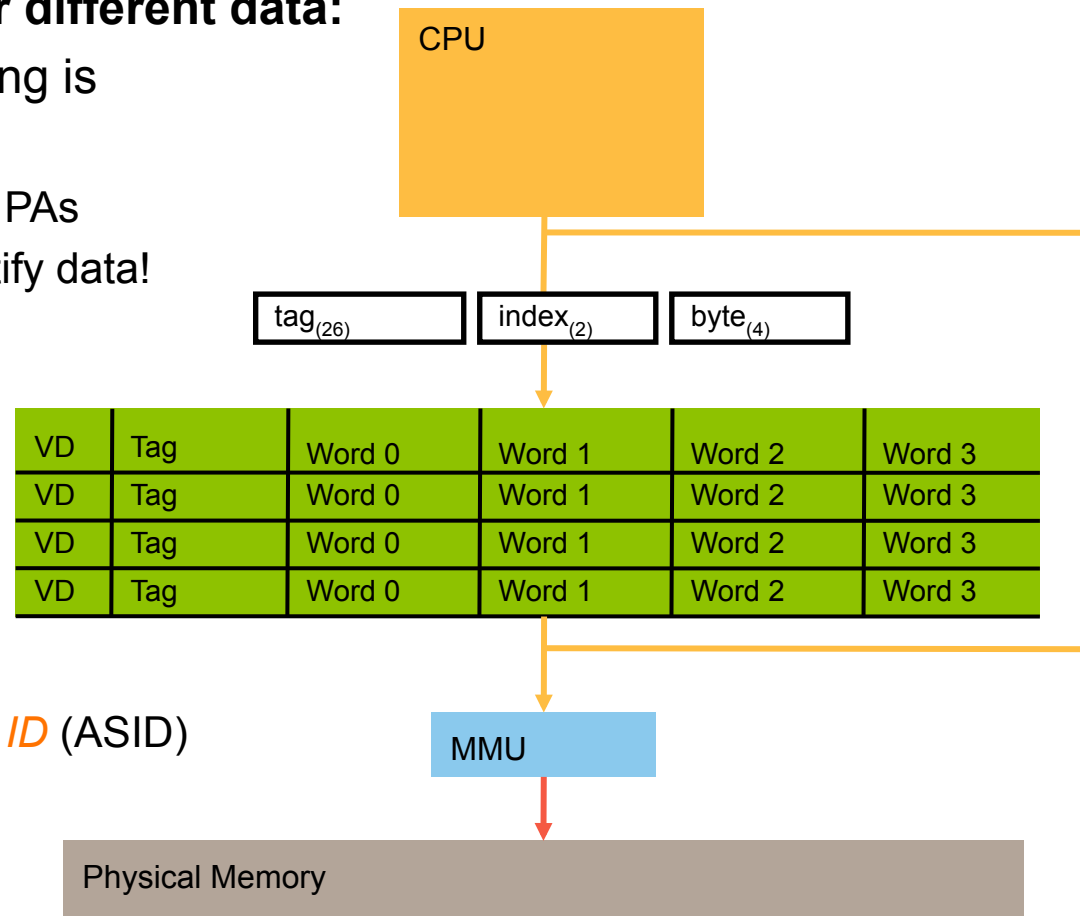


# Virtually-Indexed Cache Issues



## Homonyms — same name for different data:

- Problem: VA used for indexing is context dependent
  - same VA refers to different PAs
  - tag does not uniquely identify data!
  - wrong data is accessed!
  - an issue for most OS!
- Homonym prevention:
  - flush cache on context switch
  - force non-overlapping address-space layout
  - *tag* VA with *address-space ID* (ASID)
    - makes VAs global
  - use physical tags

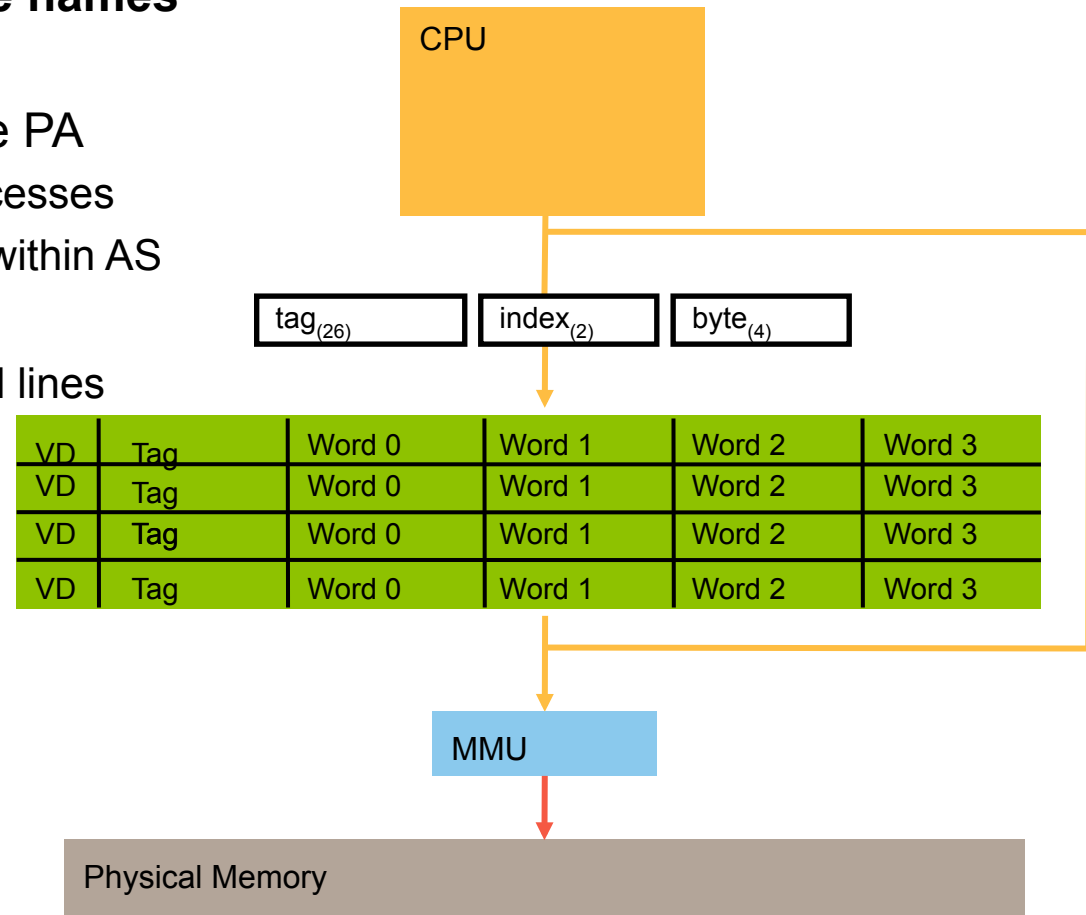


# Virtually-Indexed Cache Issues



## Synonyms (aliases) — multiple names for same data:

- Several VAs map to the same PA
  - frames shared between processes
  - multiple mappings of frame within AS
- May access stale data:
  - same data cached in several lines
  - on write, one synonym updated
  - read on other synonym returns old value!
  - physical tags don't help!
  - ASIDs don't help
- Are synonyms a problem?
  - depends on page and cache size
  - no problem for R/O data or I-caches

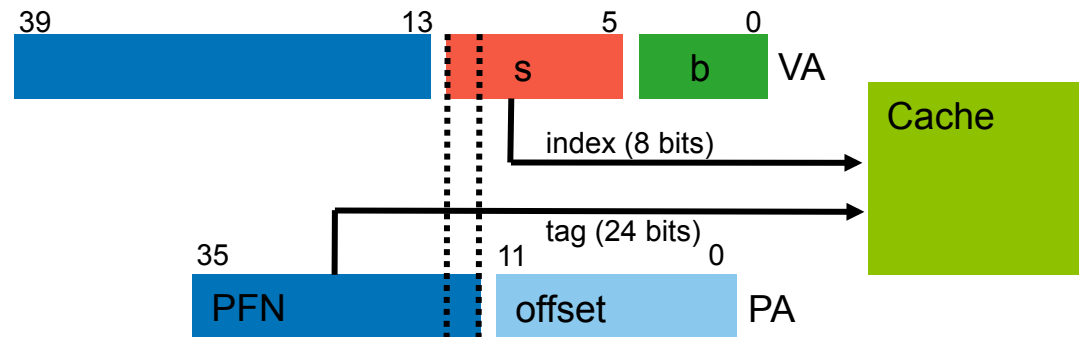


# Example: MIPS R4x00 Synonyms



→ ASID-tagged, on-chip L1 VP cache

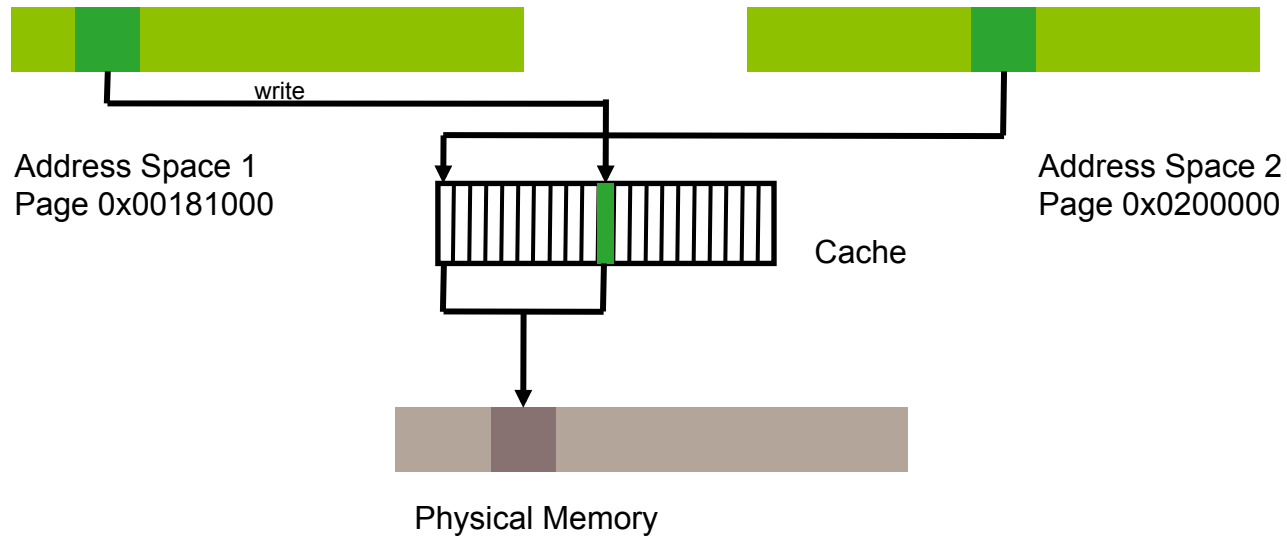
- 16KiB cache with 32B lines, 2-way set associative
- 4KiB (base) page size
- set size = 16KiB/2 = 8 KiB > page size
- overlap of tag and index bits, but come from different addresses!



→ Remember, location of data in cache determined by index

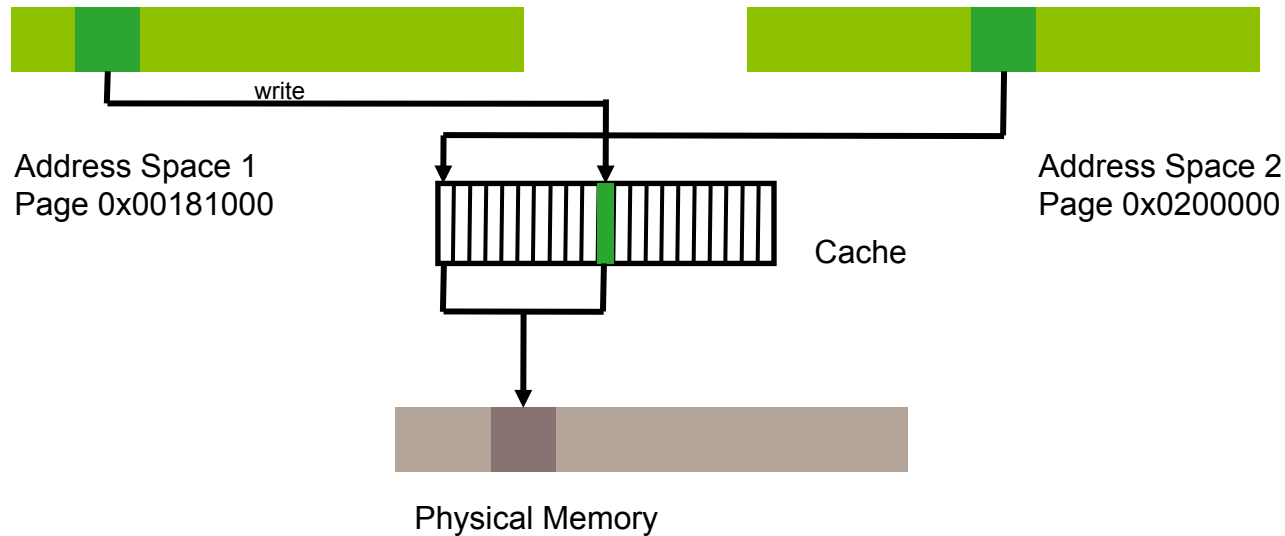
- tag only confirms whether it's a hit!
- synonym problem iff  $VA_{12} \neq VA'_{12}$
- similar issues on other processors, eg. ARM11 (set size 16KiB, page size 4KiB)

# Address Mismatch Problem: Aliasing



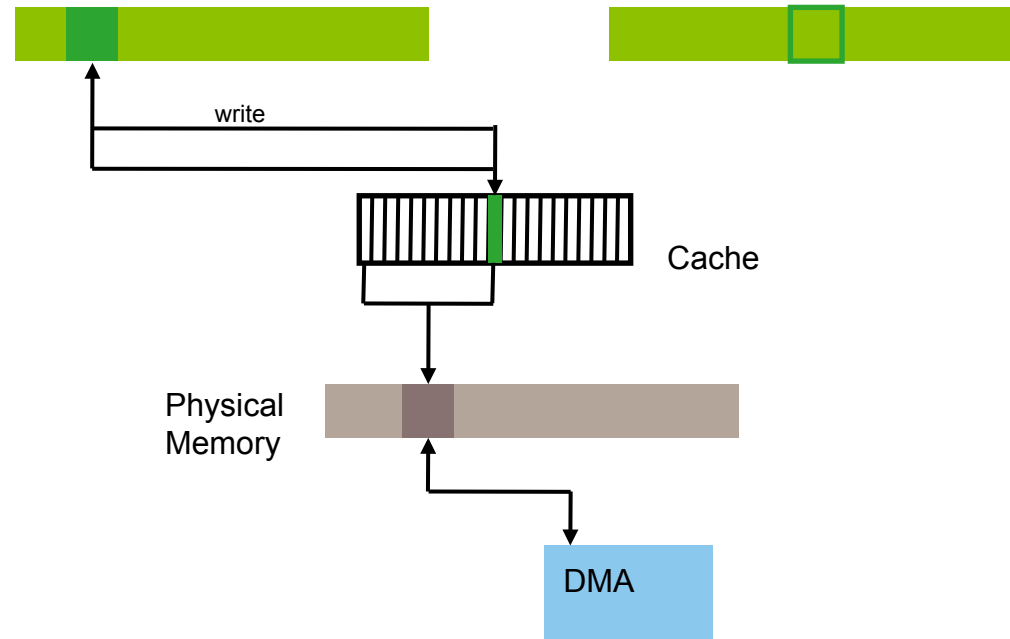
- Page aliased in different address spaces
  - $AS_1: VA_{12} = 1, AS_2: VA_{12} = 0$
- One alias gets modified
  - in a write-back cache, other alias sees stale data
  - lost-update problem

# Address Mismatch Problem: Re-Mapping



- Unmap page with a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Write to new page
  - without mismatch, new write will overwrite old (hits same cache line)
  - with mismatch, order can be reversed: “*cache bomb*”

# DMA Consistency Problem



- DMA (normally) uses physical addresses and bypasses cache
  - CPU access inconsistent with device access
  - need to flush cache before device write
  - need to invalidate cache before device read

# Avoiding Synonym Problems

---



- Hardware synonym detection
- Flush cache on context switch
  - doesn't help for aliasing *within address space*
- Detect synonyms and ensure
  - all read-only, OR
  - only one synonym mapped
- Restrict VM mapping so synonyms map to same cache set
  - e.g., R4x00: ensure that  $VA_{12} = PA_{12}$

# Summary: VV Caches

---



- ✓ Fastest (don't rely on TLB for retrieving data)
  - ✗ still need TLB lookup for protection
  - ✗ or other mechanism to provide protection
- ✗ Suffer from synonyms and homonyms
  - ✗ requires flushing on context switch
    - ✗ makes context switches expensive
    - ✗ may even be required on kernel→user switch
  - ... or guarantee of no synonyms and homonyms
- ✗ Require TLB lookup for write-back!
- Used on MC68040, i860, ARM7/ARM9/StrongARM/Xscale
- Used for I-caches on a number of architectures
  - Alpha, Pentium 4, ...

# Summary: Tagged VV Caches

---



- Add *address-space identifier* (ASID) as part of tag
- On access compare with CPU's ASID register
- ✓ Removes homonyms
  - ✓ potentially better context switching performance
  - ✗ ASID recycling still requires cache flush
- ✗ Doesn't solve synonym problem (but that's less serious)
- ✗ Doesn't solve write-back problem

# Summary: VP Caches

---



- Medium speed:
  - ✓ lookup in parallel with address translation
  - ✗ tag comparison after address translation
- ✓ No homonym problem
- ✗ Potential synonym problem
- ✗ Bigger tags (cannot leave off set-number bits)
  - ✗ increases area, latency, power consumption
- Used on most modern architectures for L1 cache

# Summary: PP Caches

---

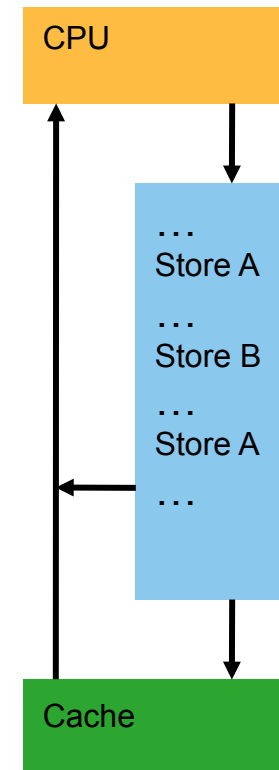


- Slowest
  - requires result of address translation before lookup starts
- No synonym problem
- No homonym problem
- Easy to manage
- If small or highly associative (all index bits come from page offset) indexing can be in parallel with address translation.
  - Potentially useful for L1 cache (used on Itanium)
- Cache can use *bus snooping to receive/supply DMA data*
- Usable as off-chip cache with any architecture
- For an in-depth coverage of caches see [Wiggins 03]

# Write Buffer



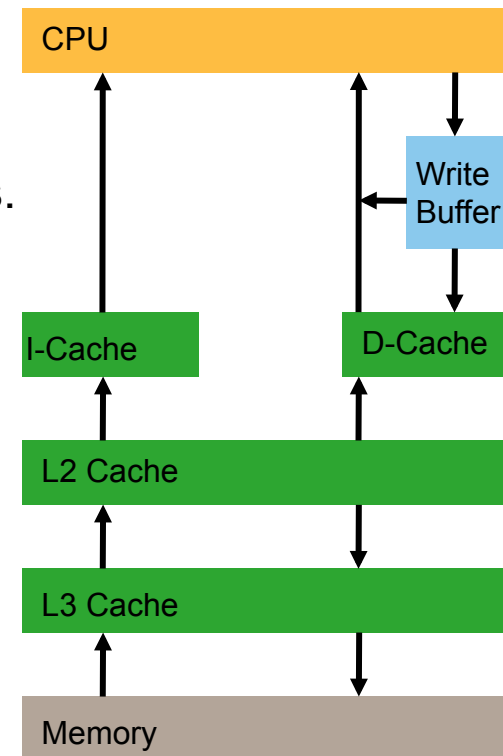
- Store operations can take a long time to complete
  - e.g. if a cache line must be read or allocated
- Can avoid stalling the CPU by buffering writes
- *Write buffer* is a FIFO *queue of incomplete stores*
  - also called *store buffer* or *write-behind buffer*
- Can also read intermediate values out of buffer
  - to service load of a value that is still in write buffer
  - avoids unnecessary stalls of load operations
- Implies that memory contents are temporarily stale
  - on a multiprocessor, CPUs see different order of writes
  - “weak store order”, to be revisited in SMP context



# Cache Hierarchy



- Hierarchy of caches to balance memory accesses:
  - small, fast, virtually indexed L1
  - large, slow, physically indexed L2–L5
- Each level reduces and clusters traffic.
- L1 typically split into instruction and data caches.
  - requirement of pipelining
- Low levels tend to be unified.
- Chip multiprocessors (multicores) often share on-chip L2, L3



# Translation Lookaside Buffer (TLB)



→ TLB is a (VV) cache for page-table entries

→ TLB can be:

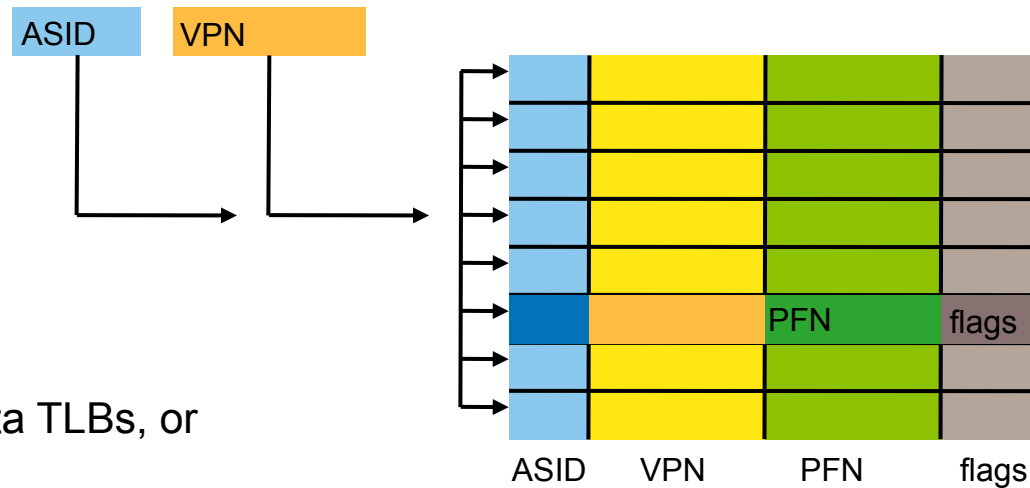
- hardware loaded, transparent to OS, or
- software loaded, maintained by OS

→ TLB can be:

- split, instruction and data TLBs, or
- unified

→ Modern high-performance architectures use a hierarchy of TLBs:

- top-level TLB is hardware-loaded from lower levels, typically split
  - transparent to OS
- second level is hardware- or software-loaded



# TLB Issues: Associativity

---



- First TLB (VAX-11/780, [Clark, Emer 85]) was 2-way associative
- Most modern architectures have fully associative TLBs
- Exceptions:
  - i486 (4-way)
  - Pentium, P6 (4-way)
  - IBM RS/6000 (2-way)
- Reasons:
  - modern architectures tend to support multiple page sizes (superpages)
    - better utilises TLB entries
  - TLB lookup done without knowing the page's base address
  - set-associativity loses speed advantage
  - superpage TLBs are fully-associative

# TLB Size (I-TLB + D-TLB)



<i>Architecture</i>	<i>TLB Size</i>	<i>Page Size</i>	<i>TLB Coverage</i>
VAX	64–256	512B	32–128KiB
ix86	32–32+64	4KiB+4MiB	128–128+256KiB
MIPS	96–128	4KiB–16MiB	384KiB–...
SPARC	64	8KiB–4MiB	512KiB–...
Alpha	32–128+128	8KiB–4MiB	256KiB–...
RS/6000	32+128	4KiB	128+512KiB
Power-4/G5	128	4KiB+16MiB	512KiB–...
PA-8000	96+96	4KiB–64MiB	
Itanium	64+96	4KiB–4GiB	

Not much growth in 20 years!

# TLB Size (I-TLB + D-TLB)

---



## TLB coverage

- Memory sizes are increasing
- Number of TLB entries are more-or-less constant
- Page sizes are growing *very slowly*
  - total amount of RAM mapped by TLB is not changing much
  - fraction of RAM mapped by TLB is shrinking dramatically
- Modern architectures have very low *TLB coverage*
- Also, many modern architectures have software-loaded TLBs
  - General increase in TLB miss handling cost
- The TLB can become a performance bottleneck