UNSW
THE UNIVERSITY OF NEW SOUTH WALES

COMP9242
Advanced Operating Systems

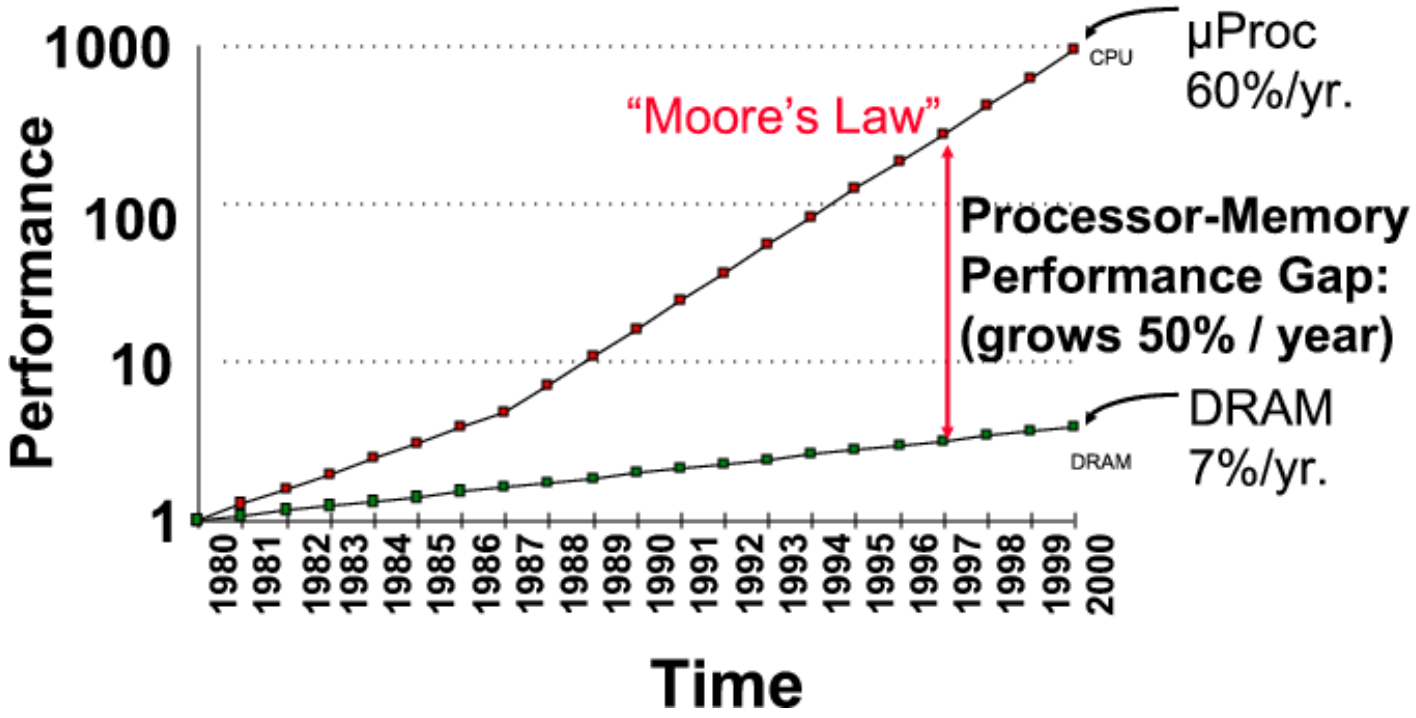
S2/2013 Week 2:
Caches:
What every OS Designer Must Know

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

NICTA Funding and Supporting Members and Partners
Australian National University
UNSW THE UNIVERSITY OF NEW SOUTH WALES
NSW GOVERNMENT
State Government Victoria
THE UNIVERSITY OF MELBOURNE
THE UNIVERSITY OF SYDNEY
Queensland Government
Griffith UNIVERSITY
QUT Queensland University of Technology
THE UNIVERSITY OF QUEENSLAND AUSTRALIA

Copyright Notice

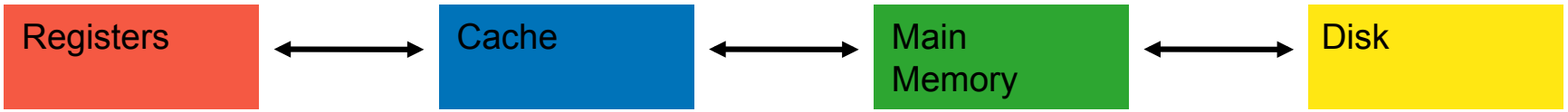
The Memory Wall



Multicore offsets stagnant per-core performance with proliferation of cores

- Basic trend is unchanged

Caching



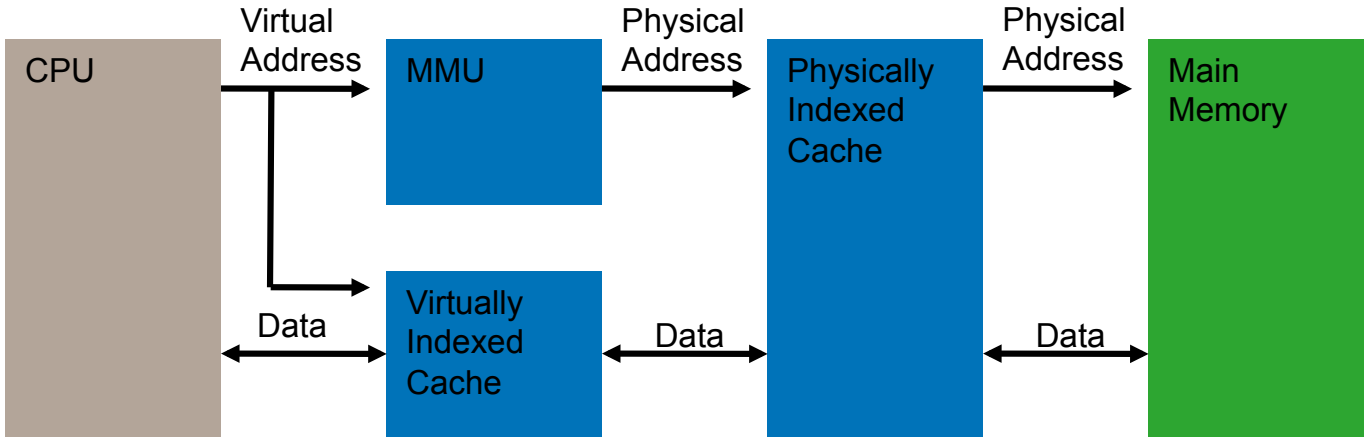
- Cache is fast (1–5 cycle access time) memory sitting between fast registers and slow RAM (10–100 cycles access time)
- Holds recently-used data or instructions to save memory accesses
- Matches slow RAM access time to CPU speed if high **hit rate** *(> 90%)*
- Is hardware maintained and (mostly) transparent 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

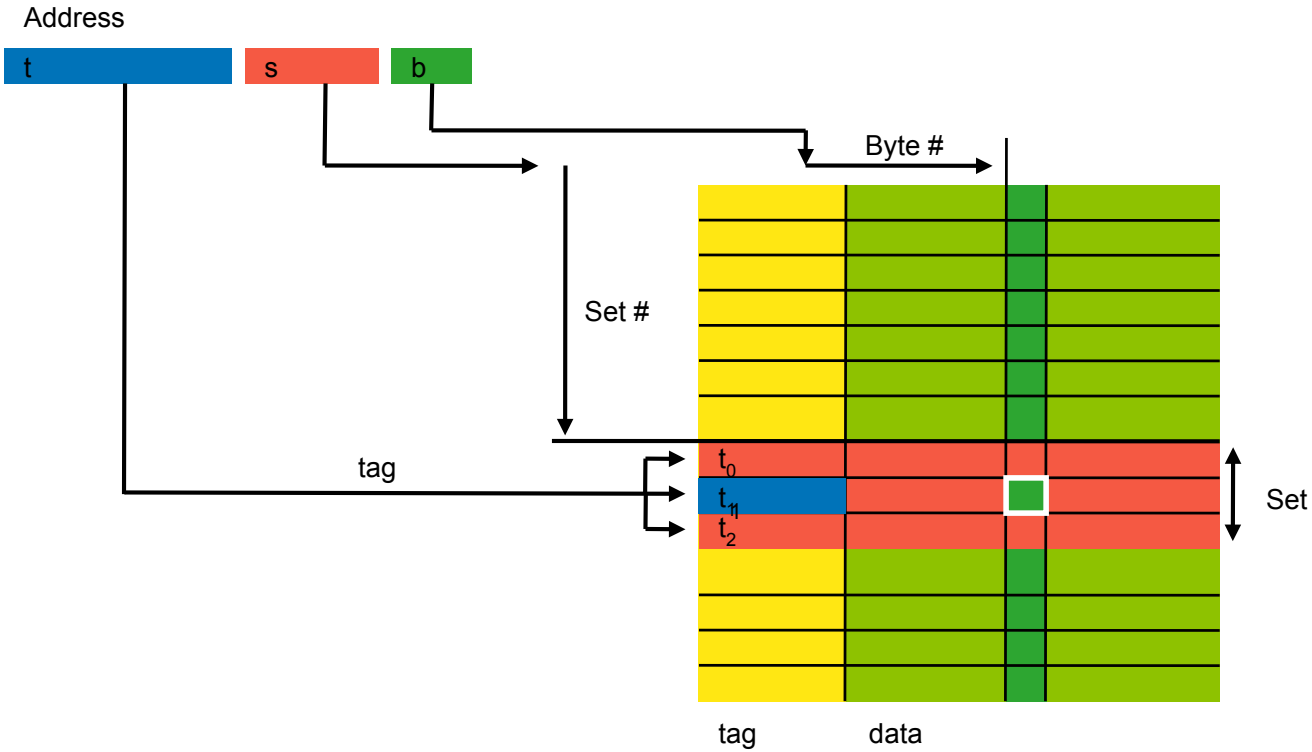
- Data transfer unit between registers and L1 cache: ≤ 1 word (1–16B)
- Cache *line* is transfer unit between cache and RAM (or lower cache)
  - typically 16–32 bytes, sometimes 128 bytes and more
- Line is also unit of storage allocation in cache
- Each line has associated control info:
  - valid bit
  - 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



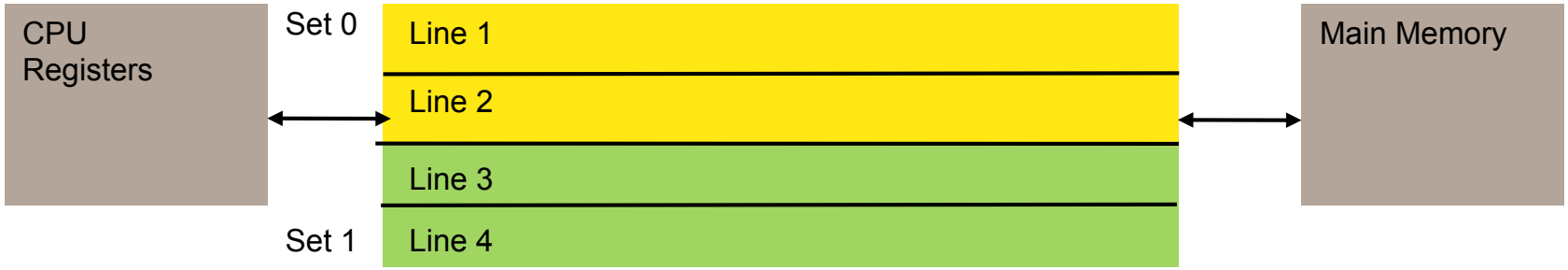
- **Virtually indexed:** looked up by *virtual address*
  - operates concurrently with address translation
- **Physically indexed:** looked up by *physical address*
  - requires result of address translation
- Usually have hierarchy: L1 (closest to core), … Ln (closest to RAM)
  - different levels may use different addresses

Cache Indexing



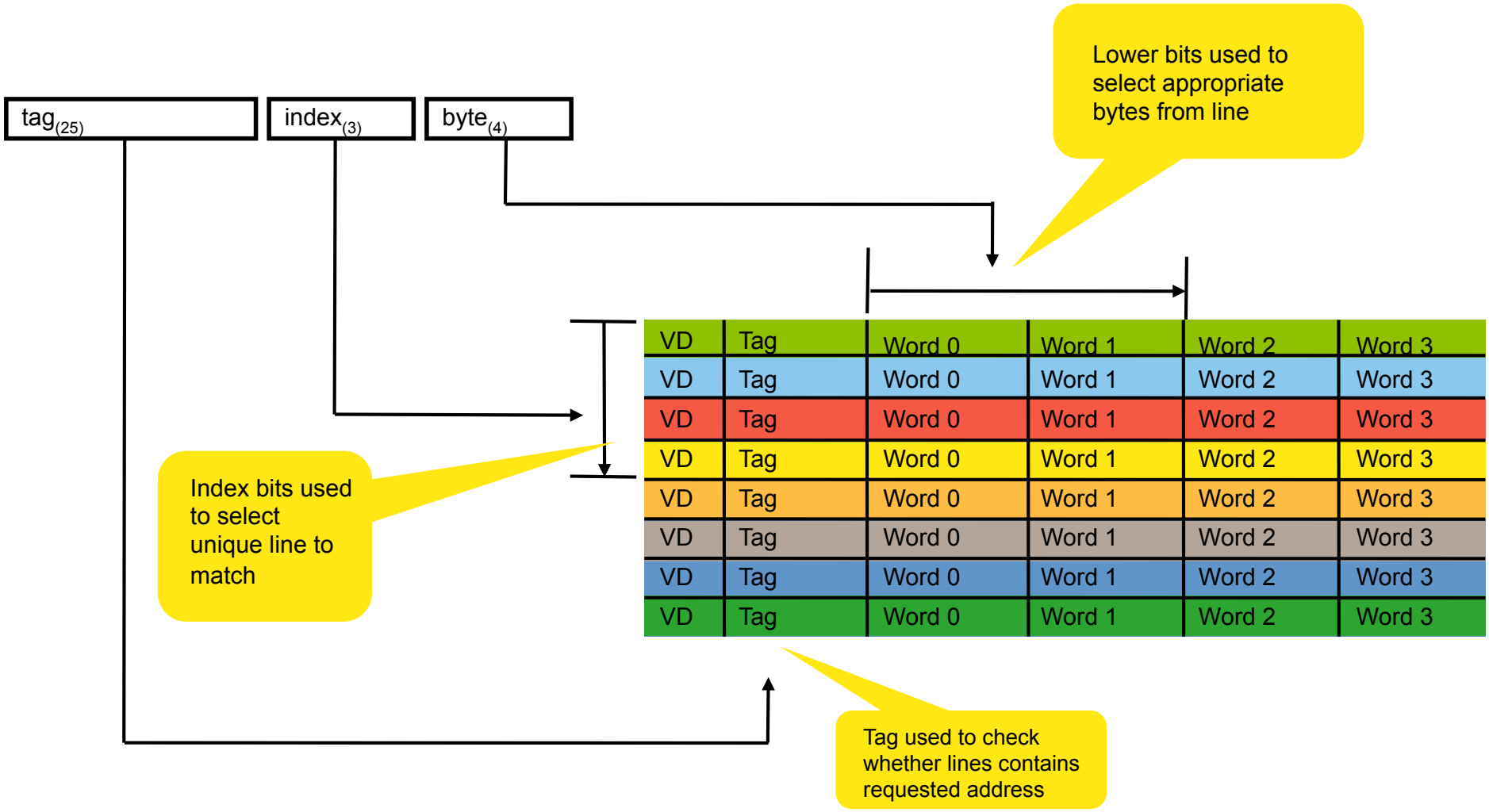
- The *tag* is used to distinguish lines of set…
- Consists of high-order bits not used for indexing

Cache Indexing

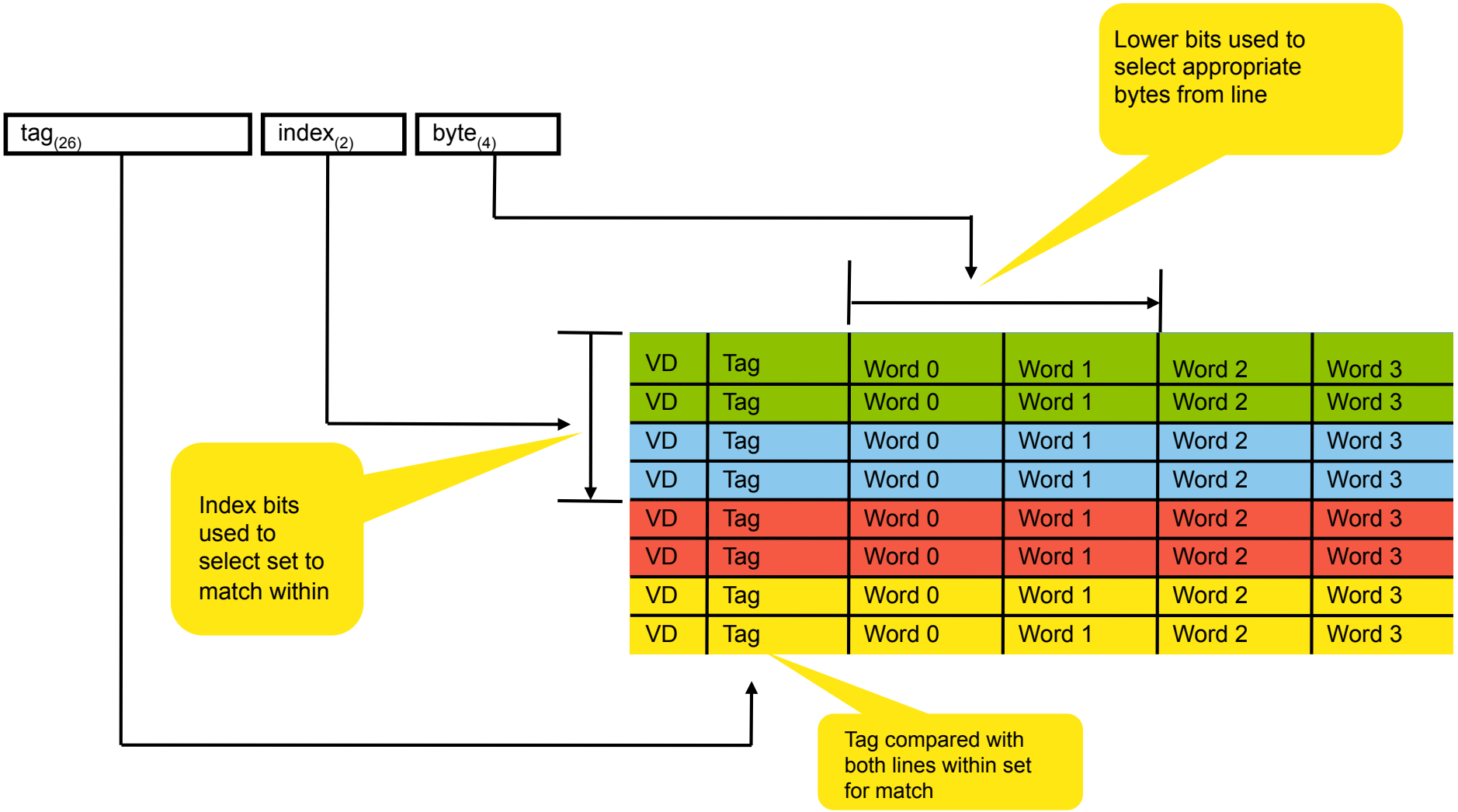


- Address hashed to produce index of *line set*
- Associative lookup of line within set
- *n* lines per set: *n-way set-associative cache*
  - typically n = 1 … 5, some embedded processors use 32-64
  - n = 1 is called direct mapped
  - n = ∞ is called fully associative (unusual for I/D caches)
- Hashing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

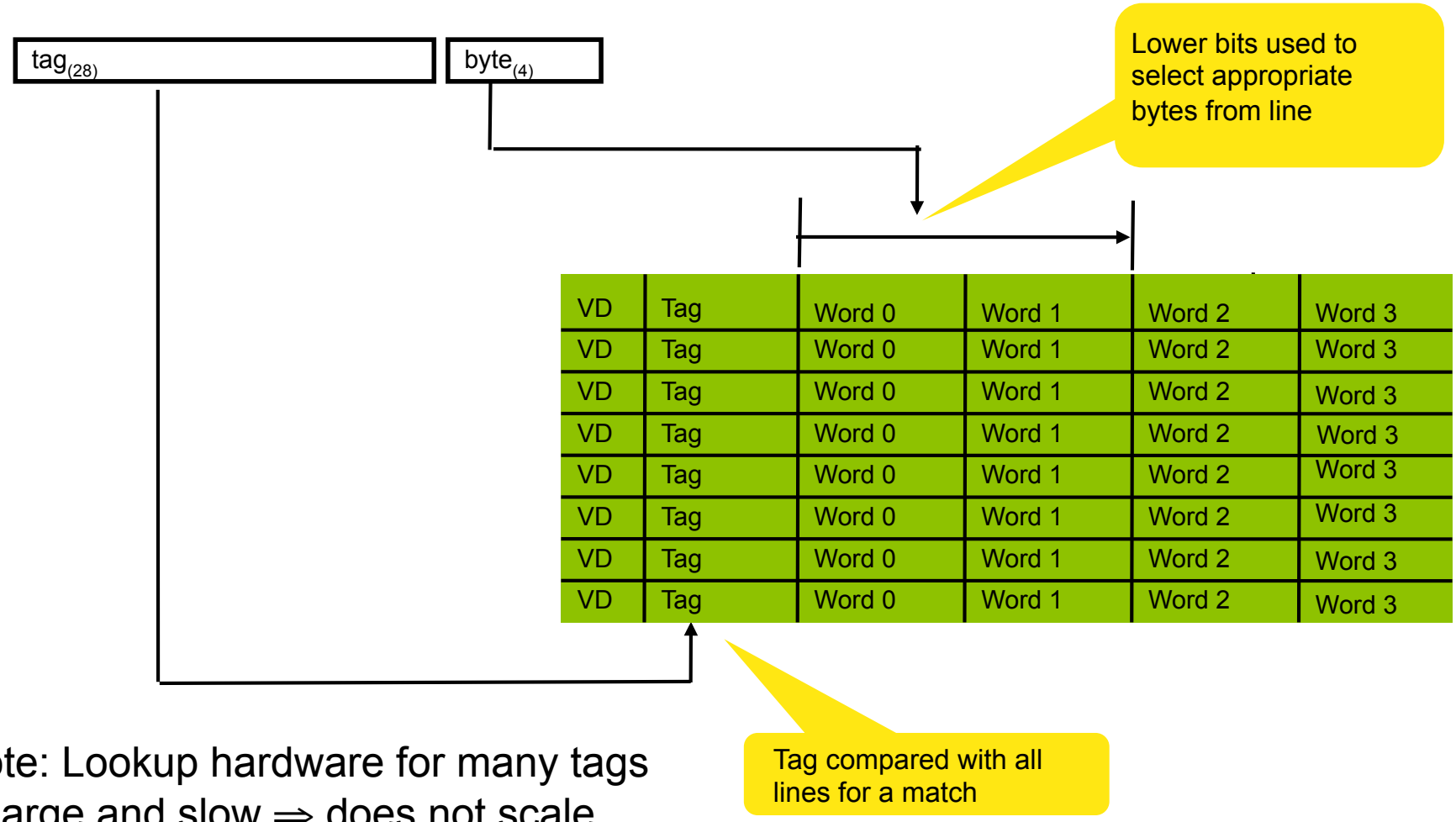
Cache Indexing: Direct Mapped



Cache Indexing: 2-Way Associative



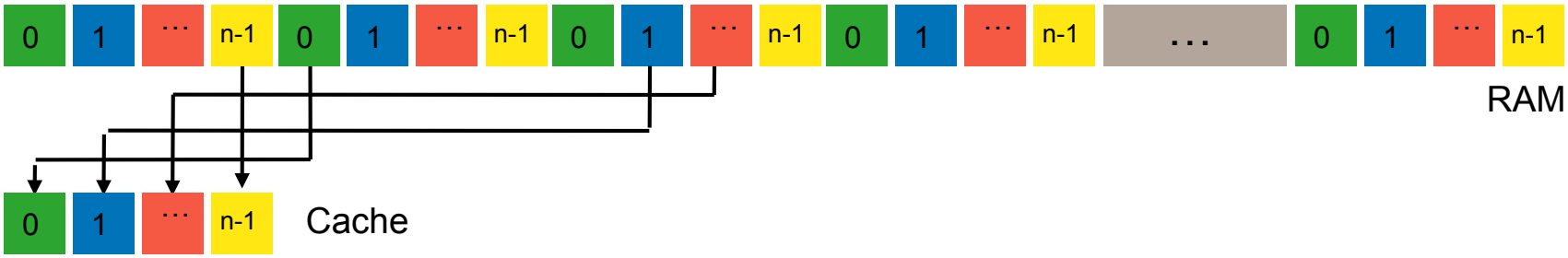
Caching Index: Fully Associative



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

Cache Mapping

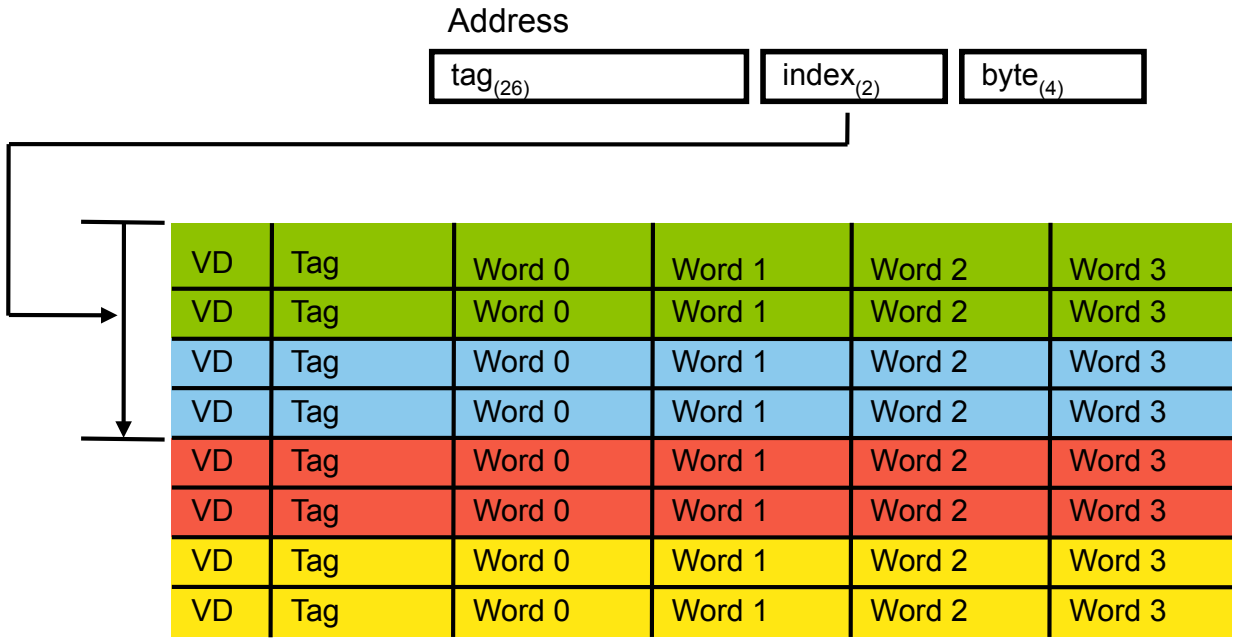
- Multiple memory locations map to the same cache line



- 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 ("the four Cs"):
  - **Compulsory miss:** data cannot be in the cache (if infinite size)
    - first access (after flush)
  - **Capacity miss:** all cache entries are in use by other data
    - would not miss on infinite-size cache
  - **Conflict miss:** all lines of the correct colour are in use by other data
    - would not miss on fully-associative cache
  - **Coherence miss:** miss forced by hardware coherence protocol

Cache Replacement Policy

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



Cache Write Policy

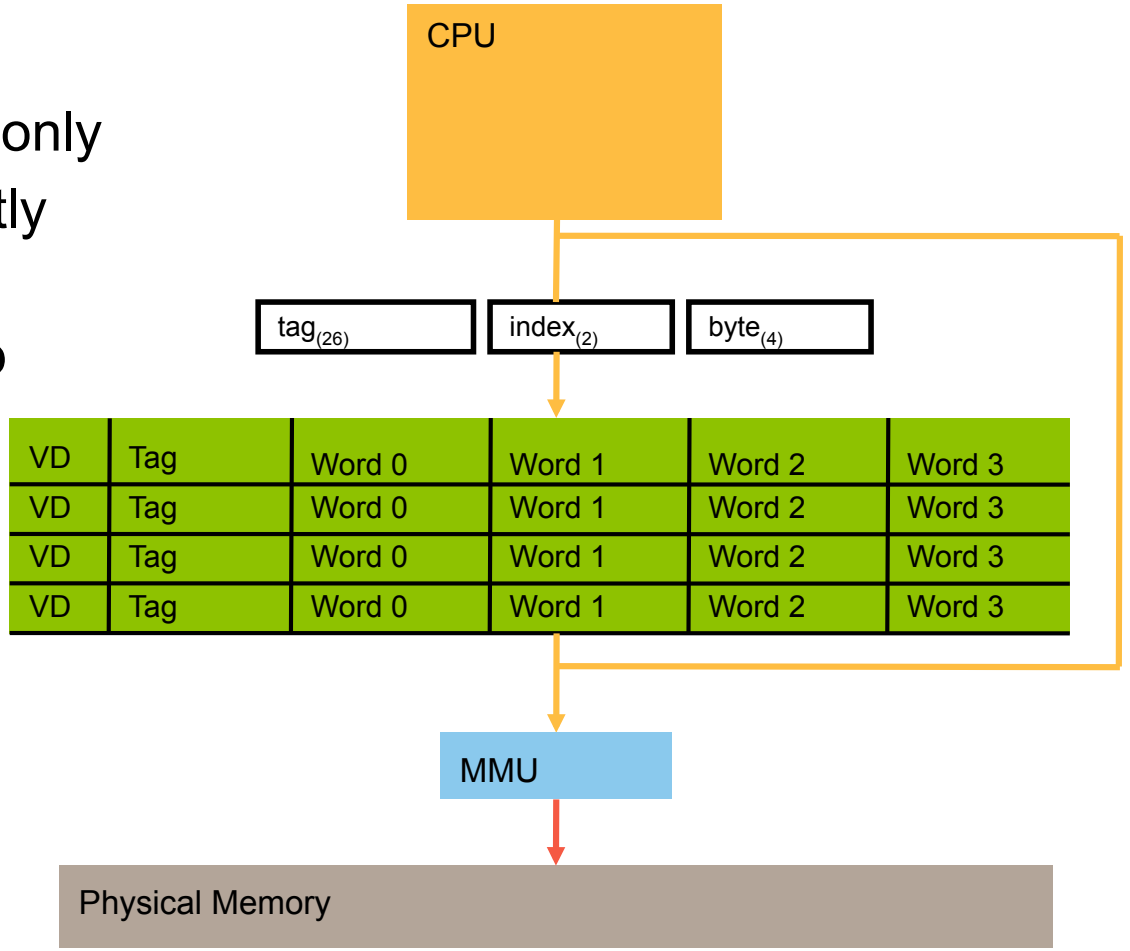
- Treatment of store operations
  - **write back:** Stores only update cache;
    memory is updated once dirty line is replaced (flushed)
    - ☑ clusters writes
    - ⌘ memory inconsistent with cache
    - ⌘ unsuitable for most multi-processor designs
  - **write through:** stores update cache and memory immediately
    - ☑ memory is always consistent with cache
    - ⌘ increased memory/bus traffic
- On store to a line not presently in cache (write miss):
  - **write allocate:** allocate a cache line and store there
    - typically requires reading line into cache first!
  - **no allocate:** store directly to memory, bypassing the cache
- Typical combinations:
  - write-back & write-allocate
  - write-through & no allocate

Cache Addressing Schemes

- For simplicity assumed so far that cache only sees one type 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 make sense only with unusual MMU designs
  - not considered any further

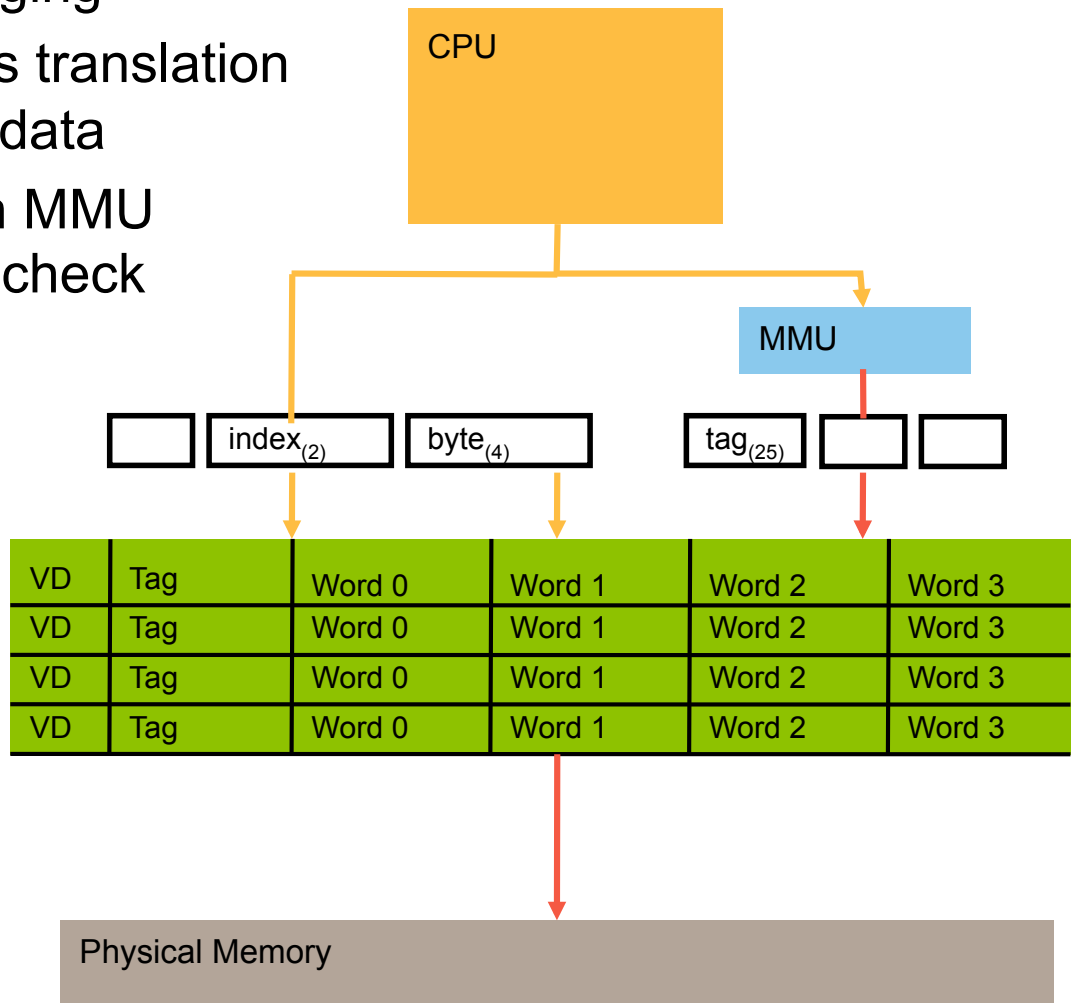
Virtually-Indexed, Virtually-Tagged Cache

- Also called *virtually-addressed cache*
- Various incorrect names in use:
  - ~~virtual cache~~
  - ~~virtual address cache~~
- Uses virtual addresses only
- Can operate concurrently with MMU
- Still needs MMU lookup for access rights
- Used for on-core L1



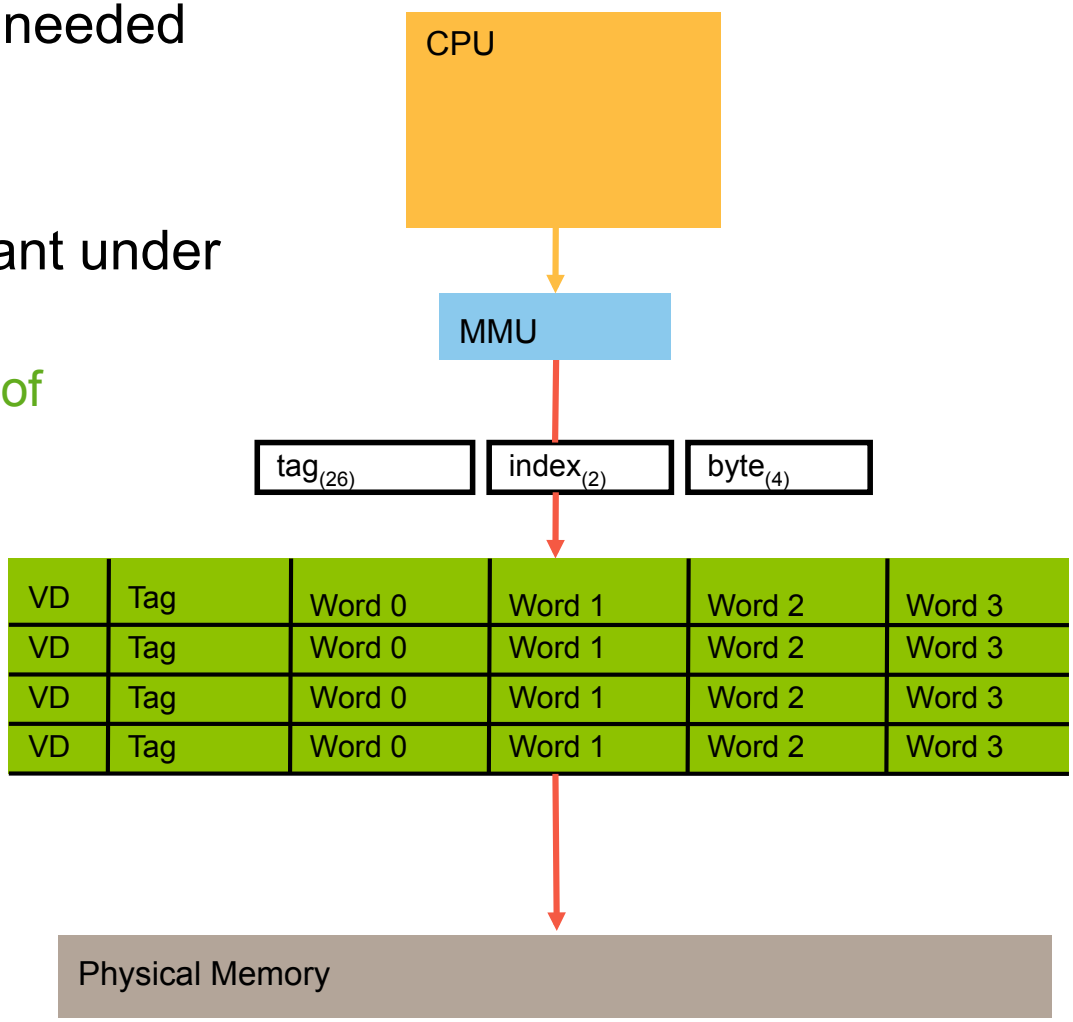
Virtually-Indexed, Physically-Tagged Cache

- Virtual address for accessing line (lookup)
- Physical address for tagging
- Needs complete address translation for looking up retrieving data
- Indexing concurrent with MMU use MMU output for tag check
- Used for on-core L1



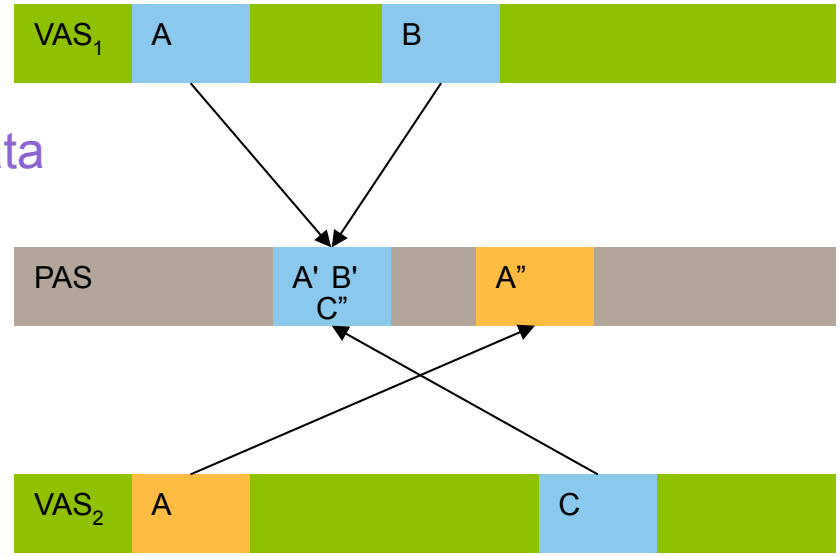
Physically-Indexed, Physically-Tagged Cache

- Only uses physical addresses
- Address translation result needed to begin lookup
- Typically used off-core
- Note: page offset is invariant under address translation
  - if index bits are a *subset* of the offset bits, PP cache lookup doesn't need MMU result!
  - VP=PP in this case fast and suitable for on-core use (L1)



Cache Issues

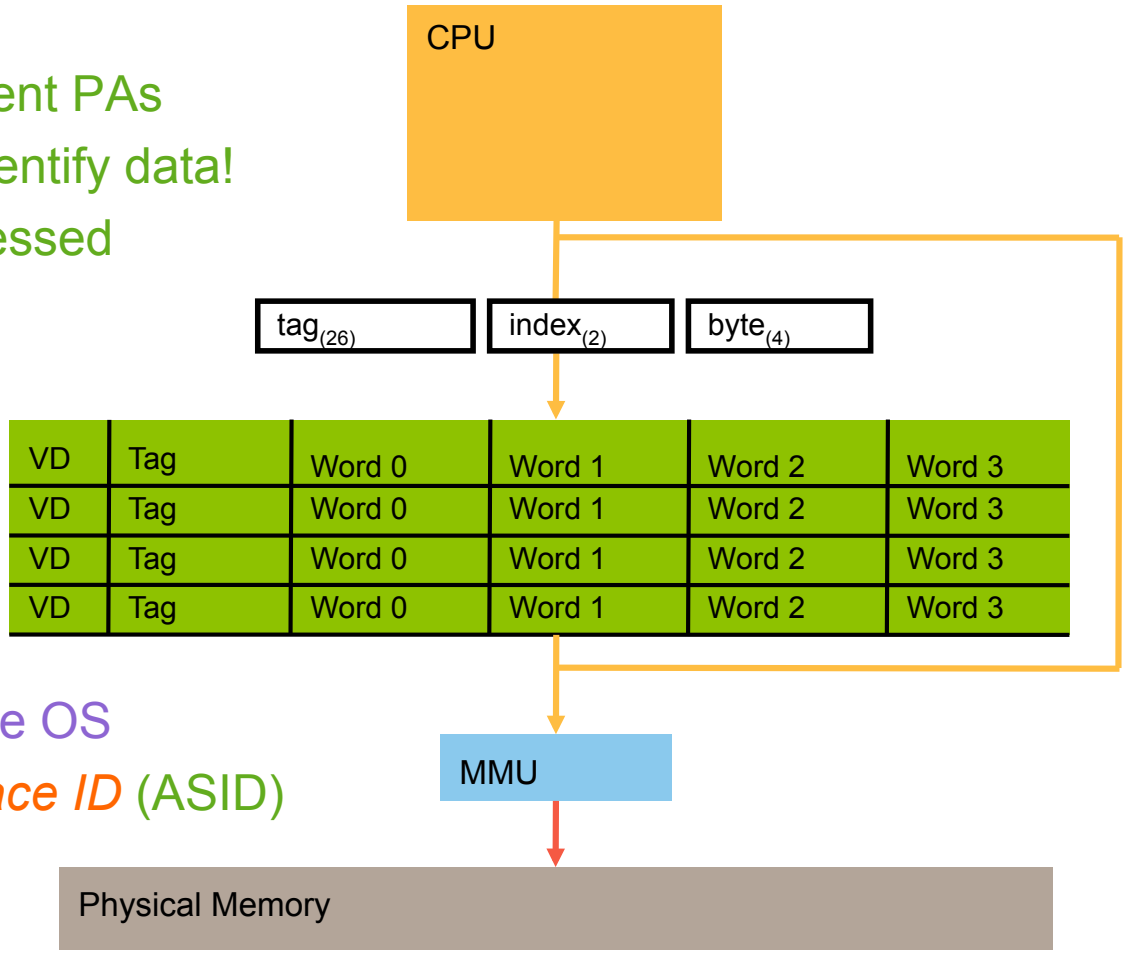
- Caches are managed by hardware transparently to software
  - OS doesn't have to worry about them, ~~right?~~ **Wrong!**
- Software-visible cache effects:
  - performance
  - *homonyms*:
    - same address, different data
    - can affect correctness!
  - *synonyms (aliases)*:
    - different address,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



Virtually-Indexed Cache Issues

Homonyms – same name for different data:

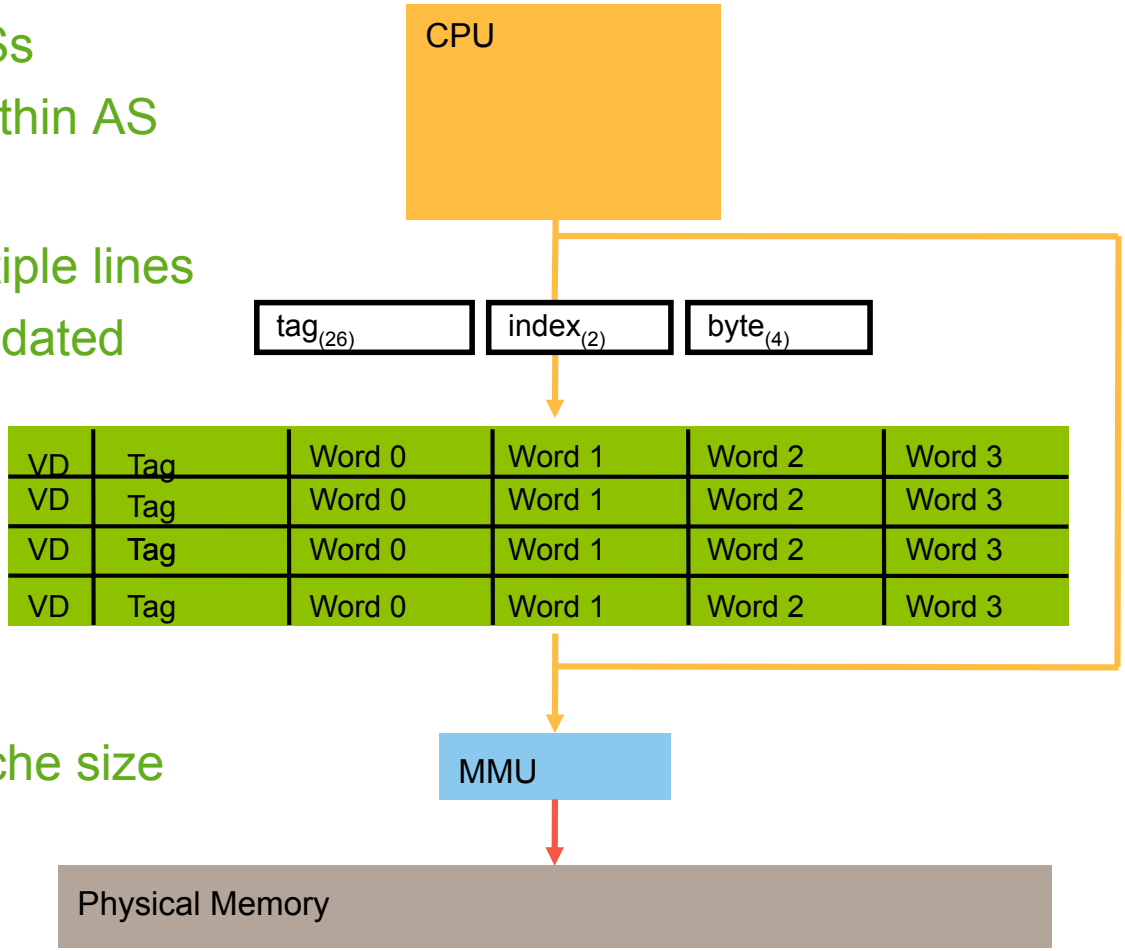
- Problem: VA used for indexing is context-dependent
  - same VA refers to different PAs
  - tag does not uniquely identify data!
  - wrong data may be accessed
  - an issue for most OSes
- Homonym prevention:
  - flush cache on each context switch
  - force non-overlapping address-space layout
    -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



Virtually-Indexed Cache Issues

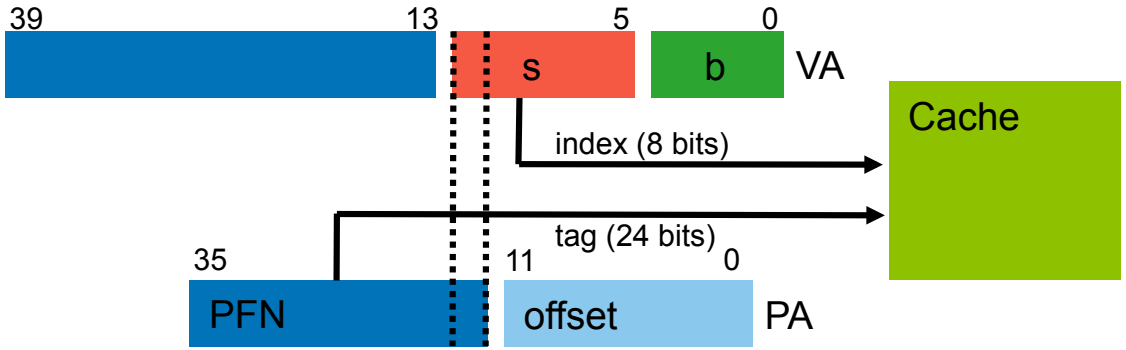
Synonyms – multiple names for same data:

- Several VAs map to the same PA
  - frame shared between ASs
  - frame multiply mapped within AS
- May access stale data!
  - same data cached in multiple 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



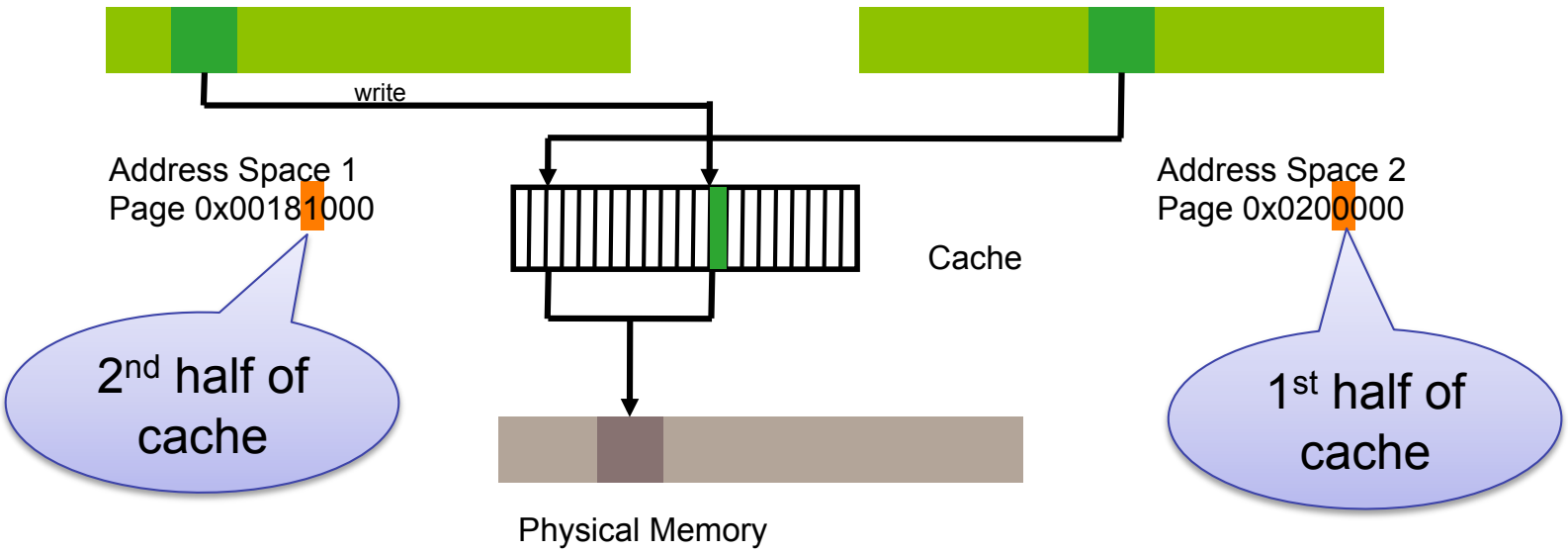
Example: MIPS R4x00 Synonyms

- ASID-tagged, on-chip VP cache
  - 16 KiB cache, 2-way set associative, 32 B line size
  - 4 KiB (base) page size
  - size/associativity = 16/2 KiB = 8 KiB > page size
    - 16 KiB / (32 B/line) = 512 lines = 256 sets ⇒ 8 index bits (12..5)
    - overlap of tag bits and index bits, but from different addresses!



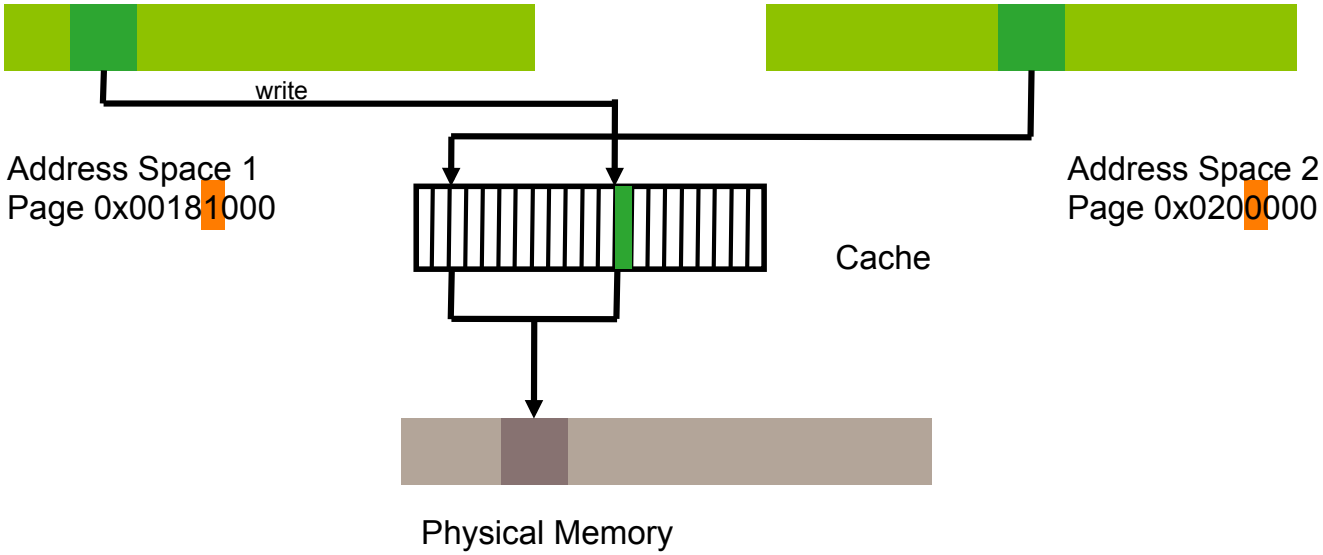
- Remember, index determines location of data in cache
  - tag only confirms hit
  - synonym problem iff $VA_{12} \neq VA'_{12}$
  - similar issues on other processors

Address Mismatch Problem: Aliasing



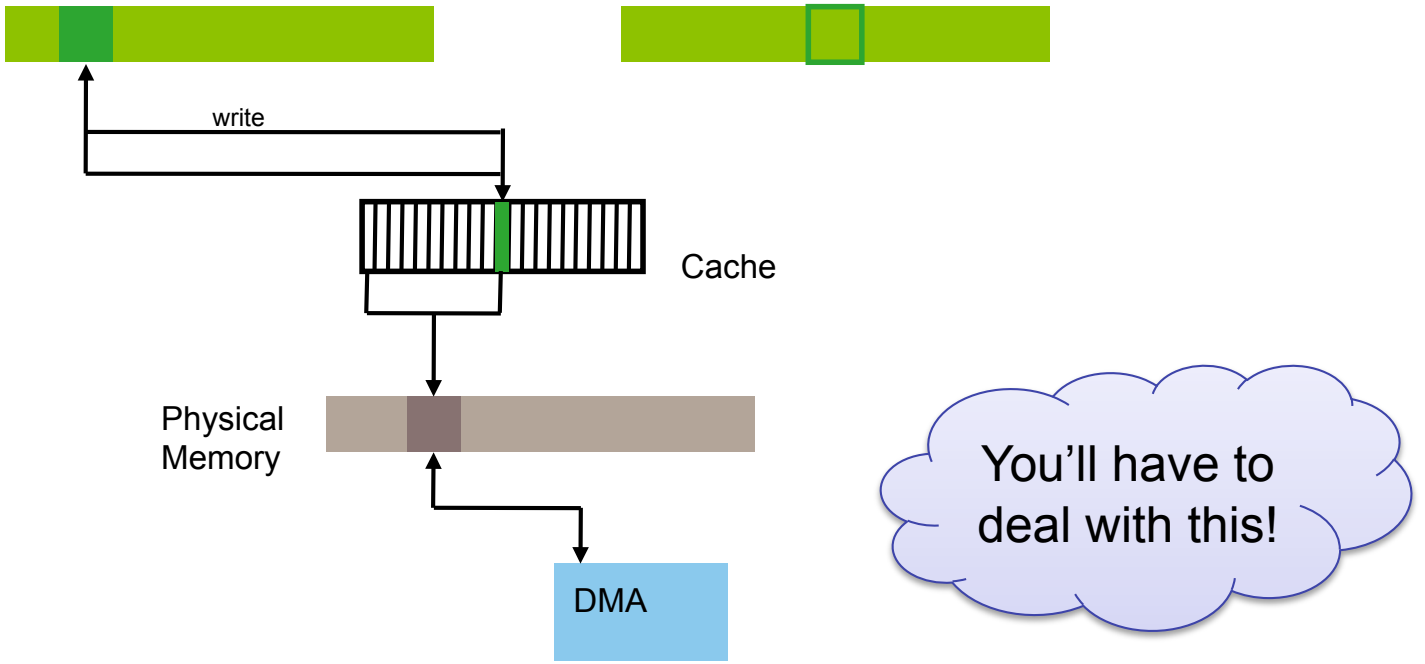
- Page aliased in different address spaces
  - AS_1: VA_{12} = 1, AS_2: VA_{12} = 0
- One alias gets modified
  - in a write-back cache, other alias sees stale data
  - lost-update problem

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

- 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
  - eg on R4x00: ensure $VA_{12} = PA_{12}$
- Hardware synonym detection
  - e.g. Cortex A9: store overlapping tag bits of both addresses & check
  - "physically"-addressed

Summary: VV Caches

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

Summary: Tagged VV Caches

- Add ASID as part of tag
- On access, compare with CPU's ASID register

☑ Removes homonyms

- ☑ potentially better context-switching performance
- ⌘ ASID recycling still needs flush

⌘ Doesn't solve synonym problem (but that's less severe)

⌘ 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

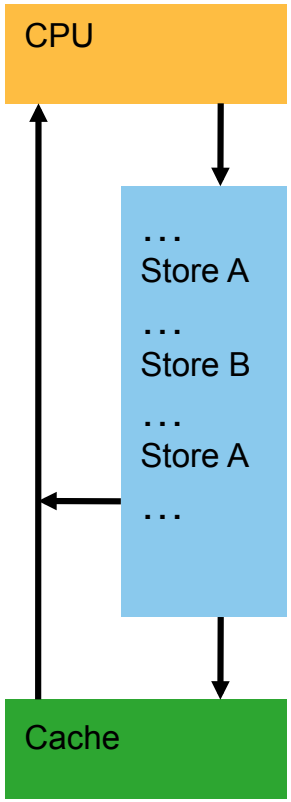
Summary: PP Caches

- ⌘ Slowest
  - ⌘ requires result of address translation before lookup starts
- ☑ No synonym problem
- ☑ No homonym problem
- ☑ Easy to manage
- ☑ If small or highly associative index can be in parallel with translation
  - all index bits come from page offset
  - combines advantages of VV and PP cache
  - useful for con-core L1 cache (Itanium, Core i7)
- ☑ Cache can use *bus snooping* to receive/supply DMA data
- ☑ Usable as post-MMU cache with any architecture

For an in-depths coverage see [Wiggins 03]

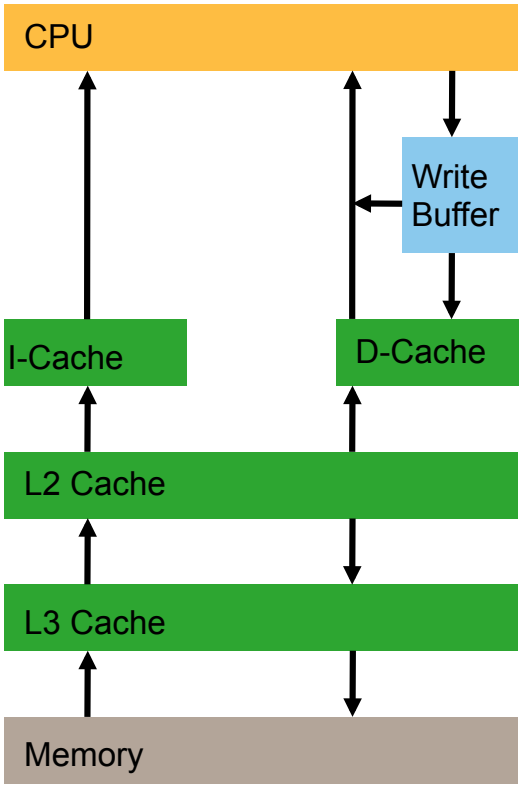
Write Buffer

- Store operations can take a long time to complete
  - eg 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 lead 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

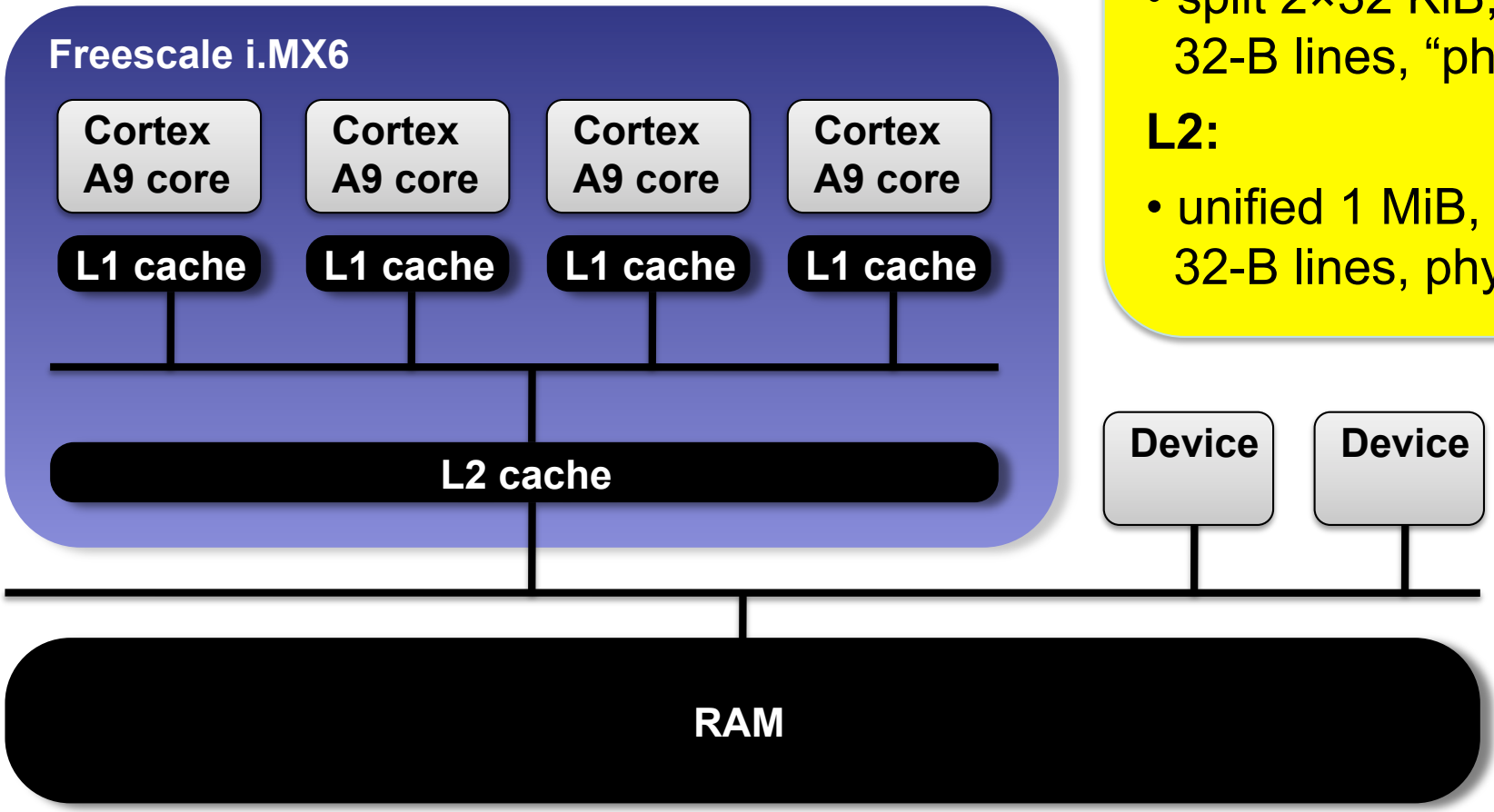


Cache Hierarchy

- 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 I- and D-caches
  - “Harvard architecture”
  - requirement of pipelining
- Other levels tend to be unified
- Chip multiprocessors often share on-chip L2, L3



Sabre (Cortex A9) System Architecture



L1:

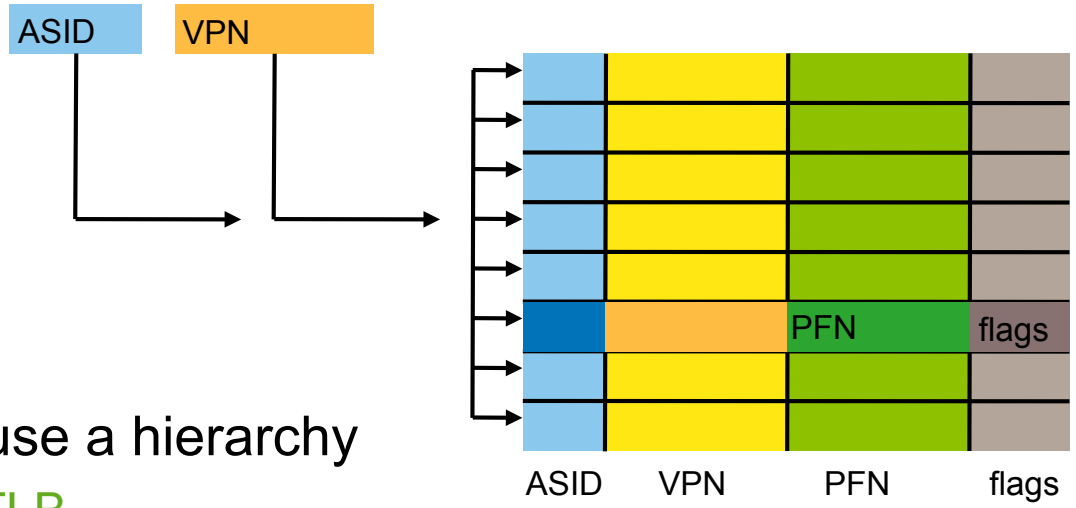
- split 2×32 KiB, 4-way, 32-B lines, “physical”

L2:

- unified 1 MiB, 16-way 32-B lines, physical

Translation Lookaside Buffer (TLB)

- TLB is a (VV) cache for page-table entries
- TLB can be
  - hardware loaded, transparent to OS
  - software loaded, maintained by OS
- TLB can be:
  - split: I- and D-TLBs
  - unified
- Modern architectures use a hierarchy
  - tiny (1-4 entries) L1 TLB
    - typically split, hardware loaded
  - larger (64-128 entries) L2 TLB
    - unified, hard- 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:
  - Intel x86: 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
- x86 uses separate L1 TLBs for each page size
  - 1 each I-TLB and D-TLB for 4 KiB and 2/4 MiB (4 L1 TLBs)
  - unified L2 TLB
  - all 4-way associative

TLB Size (I-TLB + D-TLB)

Architecture	*TLB Size*	*Page Size*	*TLB Coverage*
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 roughly constant
- Page sizes are steady
  - 4 KiB, although larger on SPARC, Alpha
- Consequences:
  - 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 64-bit RISC architectures have software-loaded TLBs
  - general increase in TLB miss handling cost
- The TLB can become a bottleneck