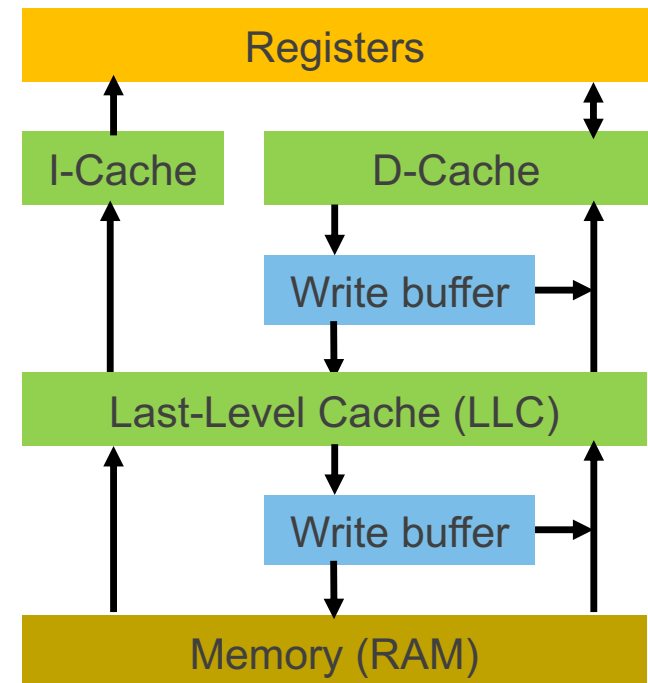


2021 T2 Week 03 Part 1

Caches:

What Every OS Designer Must Know

@GernotHeiser



Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

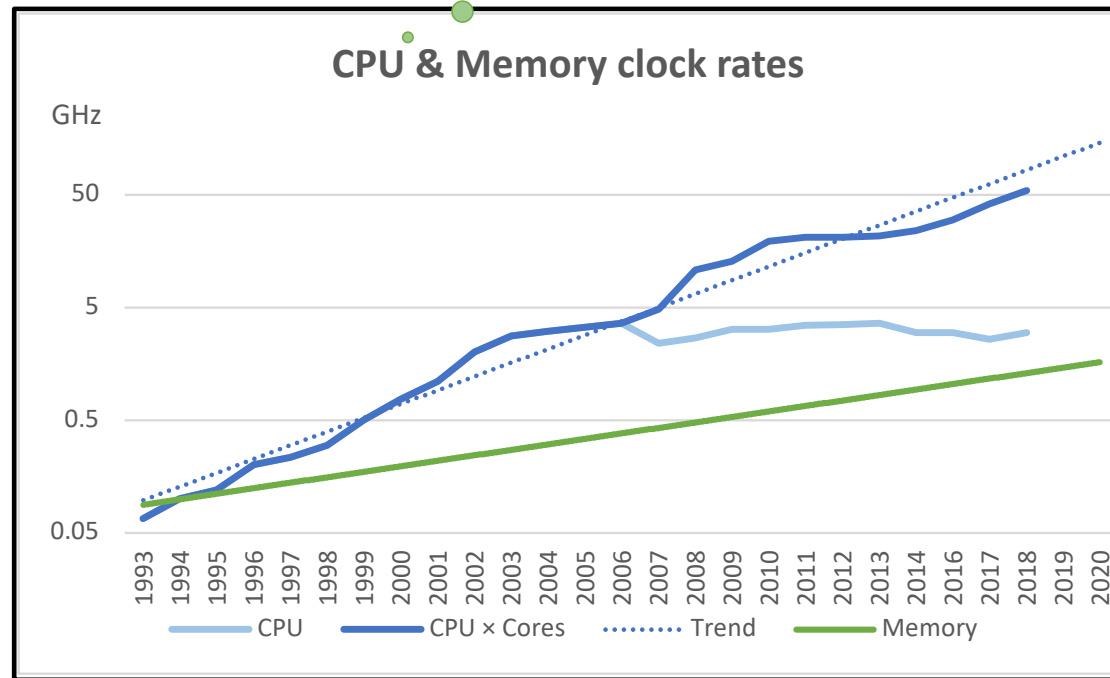
- You are free:
 - to share—to copy, distribute and transmit the work
 - to remix—to adapt the work
- under the following conditions:
 - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

“Courtesy of Gernot Heiser, UNSW Sydney”

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

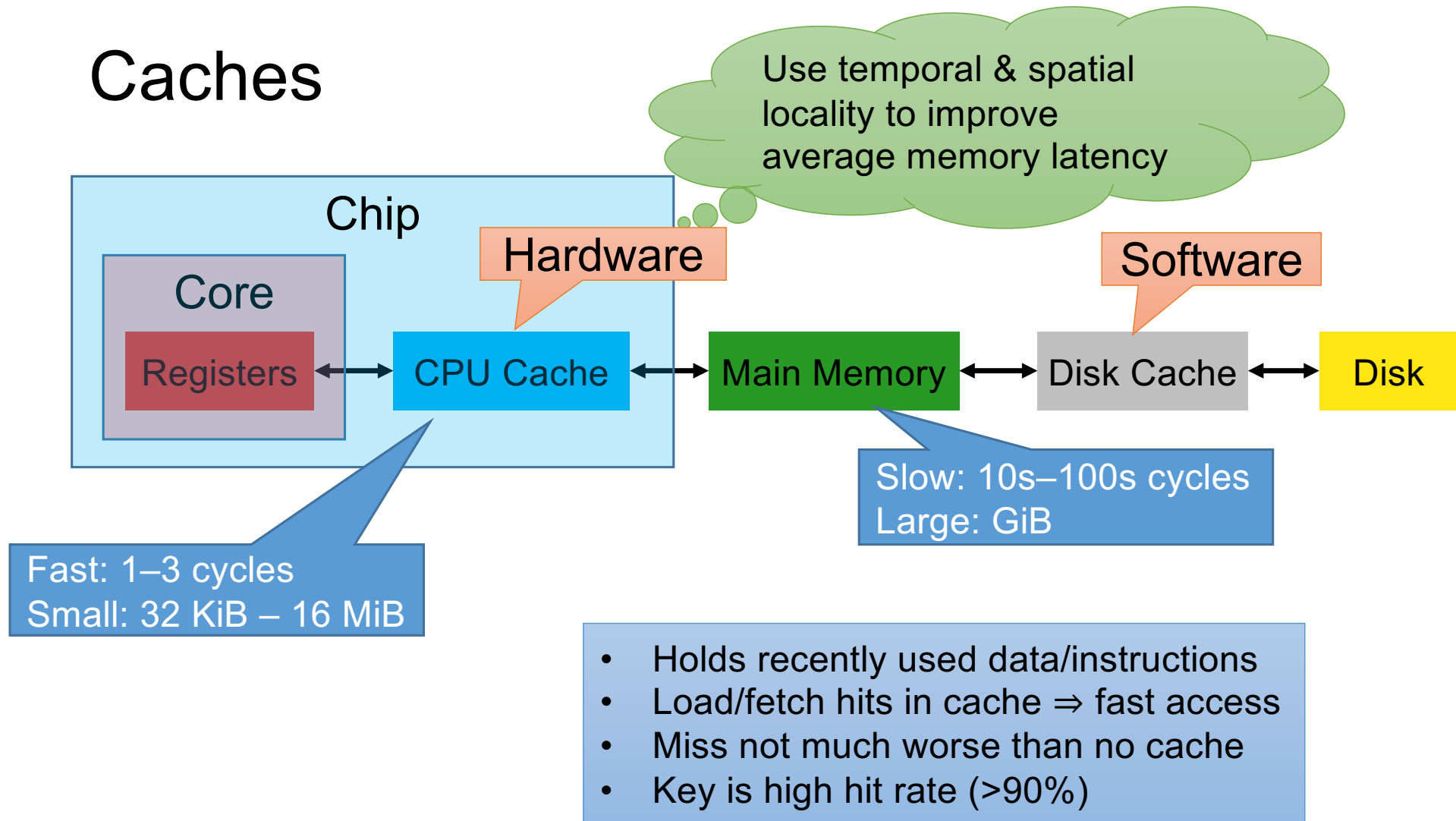
The Memory Wall

Clock rates of Intel processors



Speed gap still widens by approx 18% per year!

Caches



Cache Organisation: Unit of Data Transfer



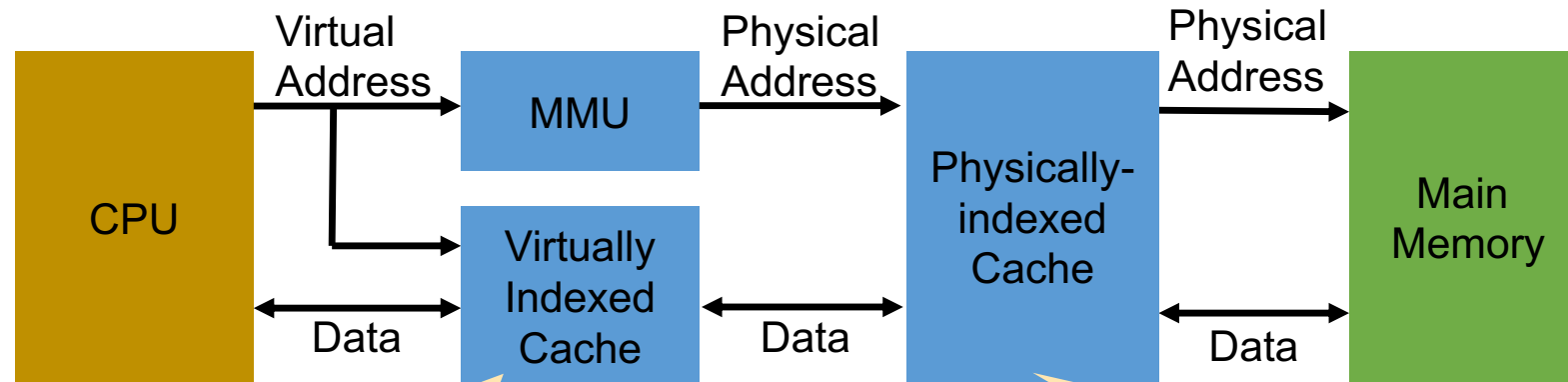
Line is also unit of allocation, holds data and

- valid bit
- modified (dirty) bit
- tag
- access stats (for replacement)

Reduce memory transactions:

- Reads – locality
- Writes – clustering

Cache Access



- Looked up by *virtual address*
- Operates concurrently with address translation

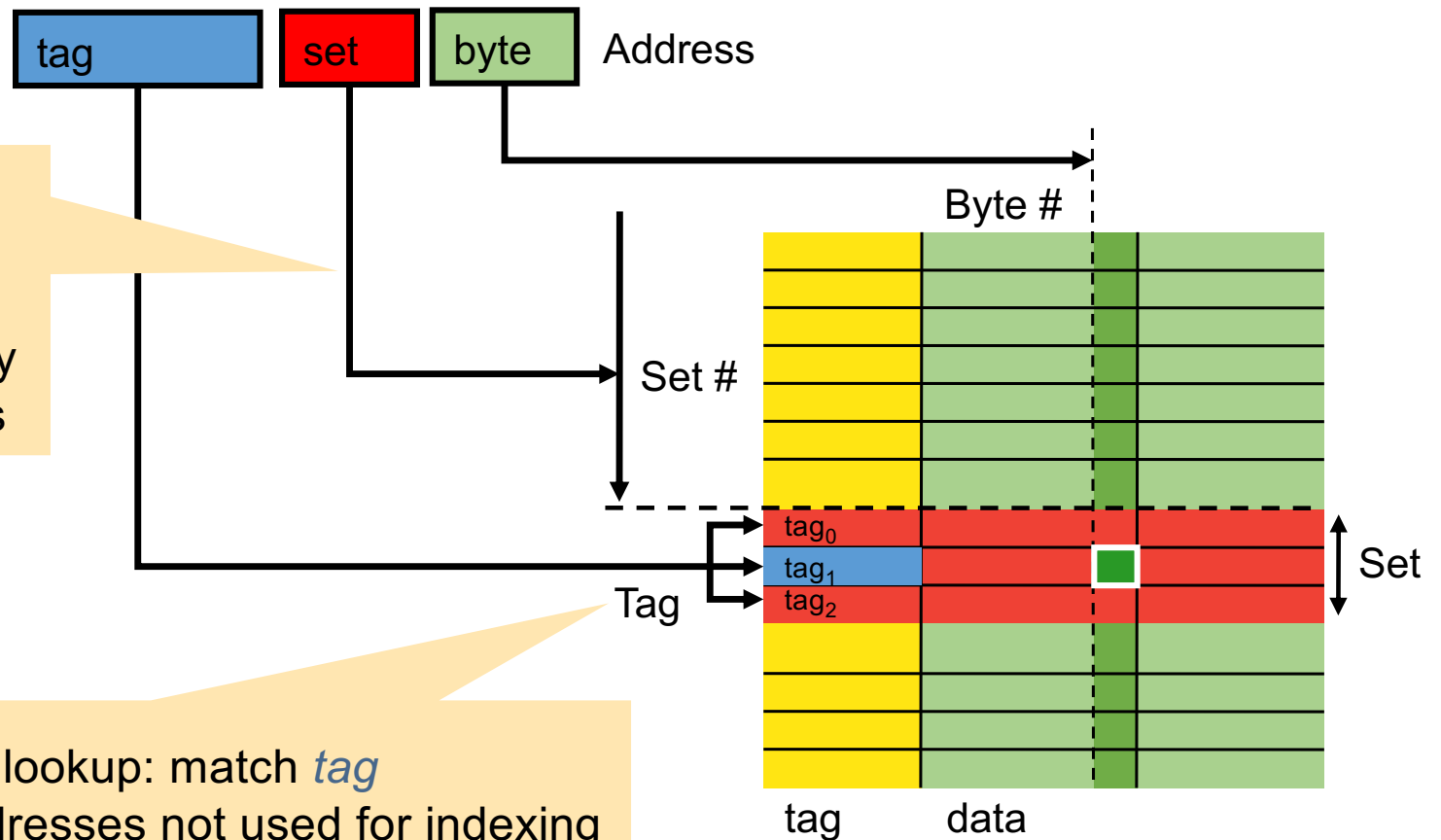
- Looked up by physical address
- Requires result of address translation

Usually a hierarchy: L1, L2, ..., LLC

- L1 closest to CPU
- LLC: last-level cache
- Only L1 may be virtually addressed

Cache Indexing

- Address hashed to produce set index
- Hashing must be simple (HW) – usually just the low-order bits



- Within set associate lookup: match *tag*
- Tag = high-order addresses not used for indexing

Cache Indexing

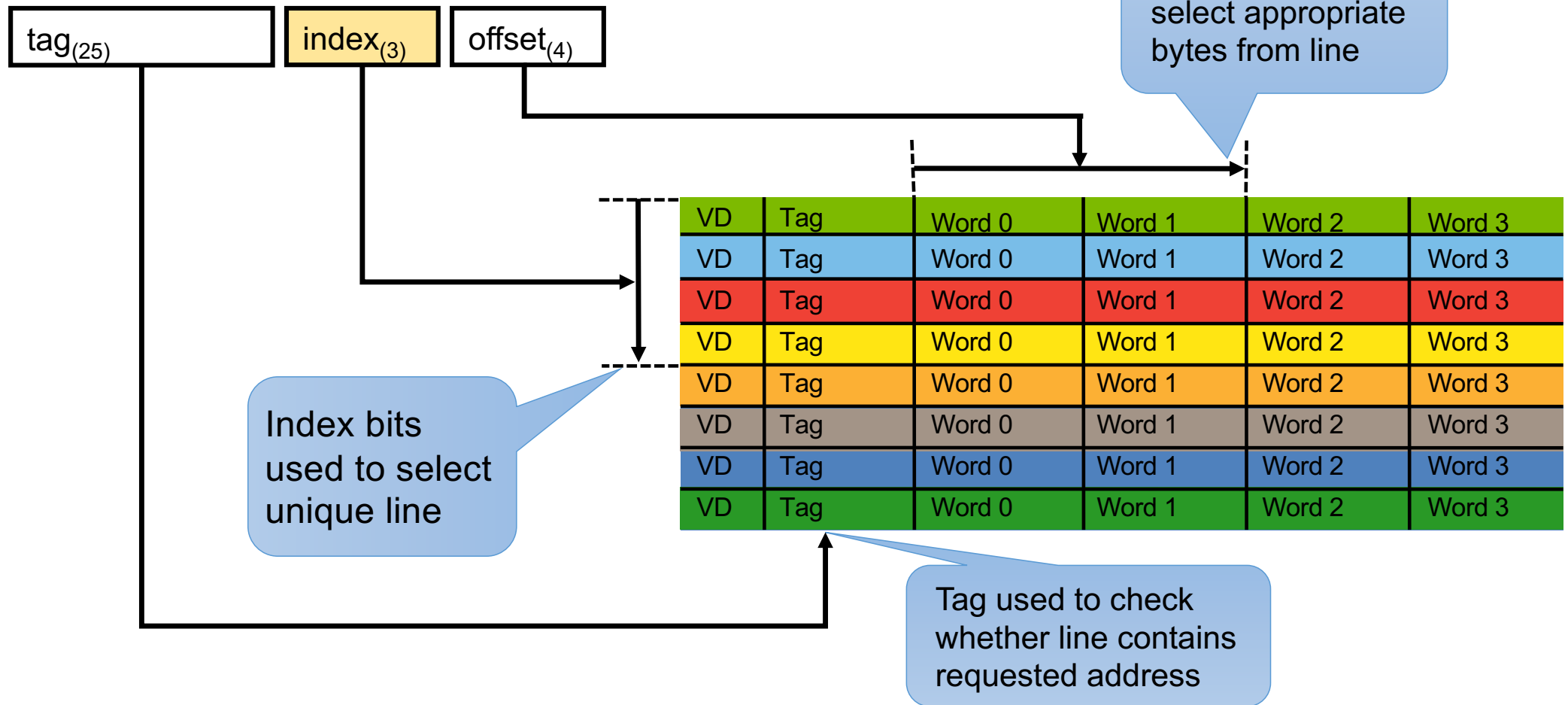


Many conflicts
⇒ low hit rate

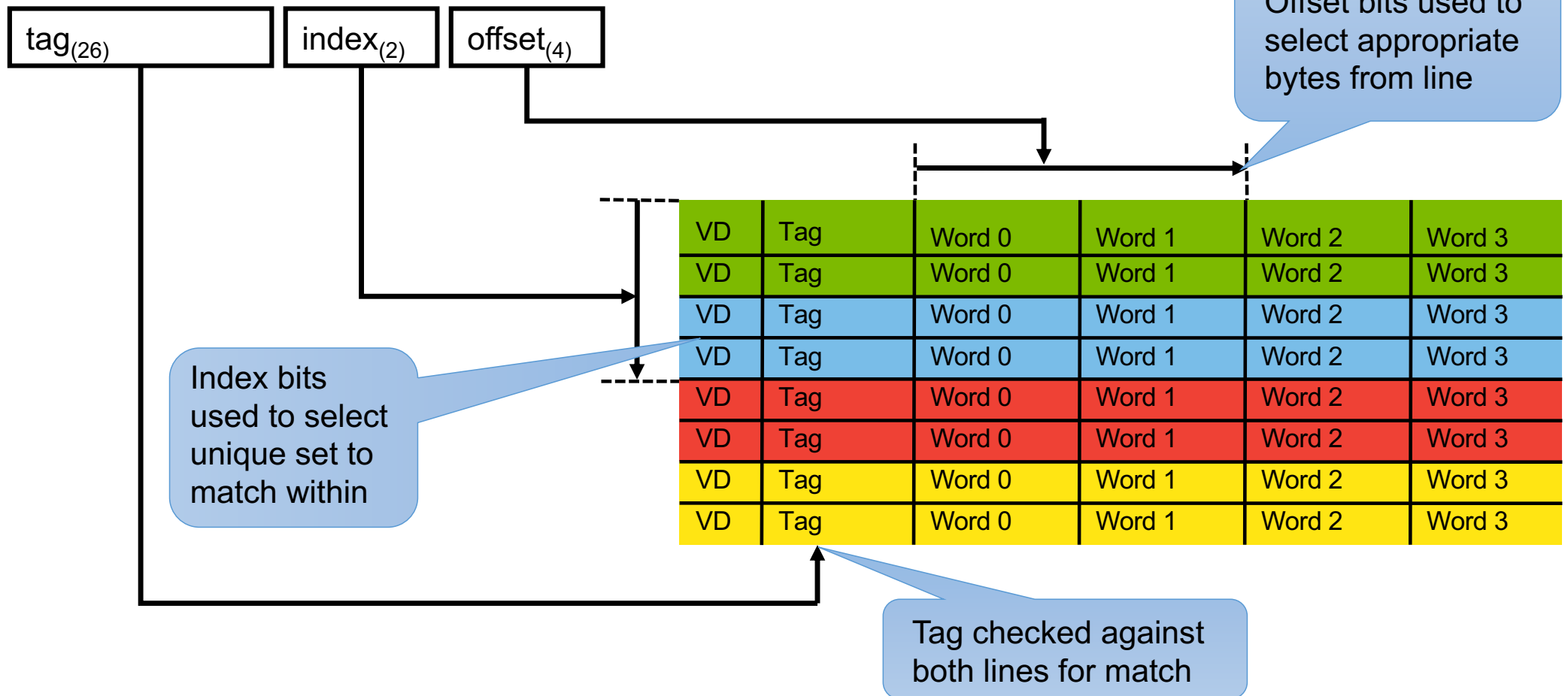
- n lines per set: n -way set-associative cache
- $n = 1$: *direct mapped*
- $2 \leq n < \# \text{ lines}$: *set associative*
- $n = \# \text{ lines}$: *fully associative*

Slow & power-hungry

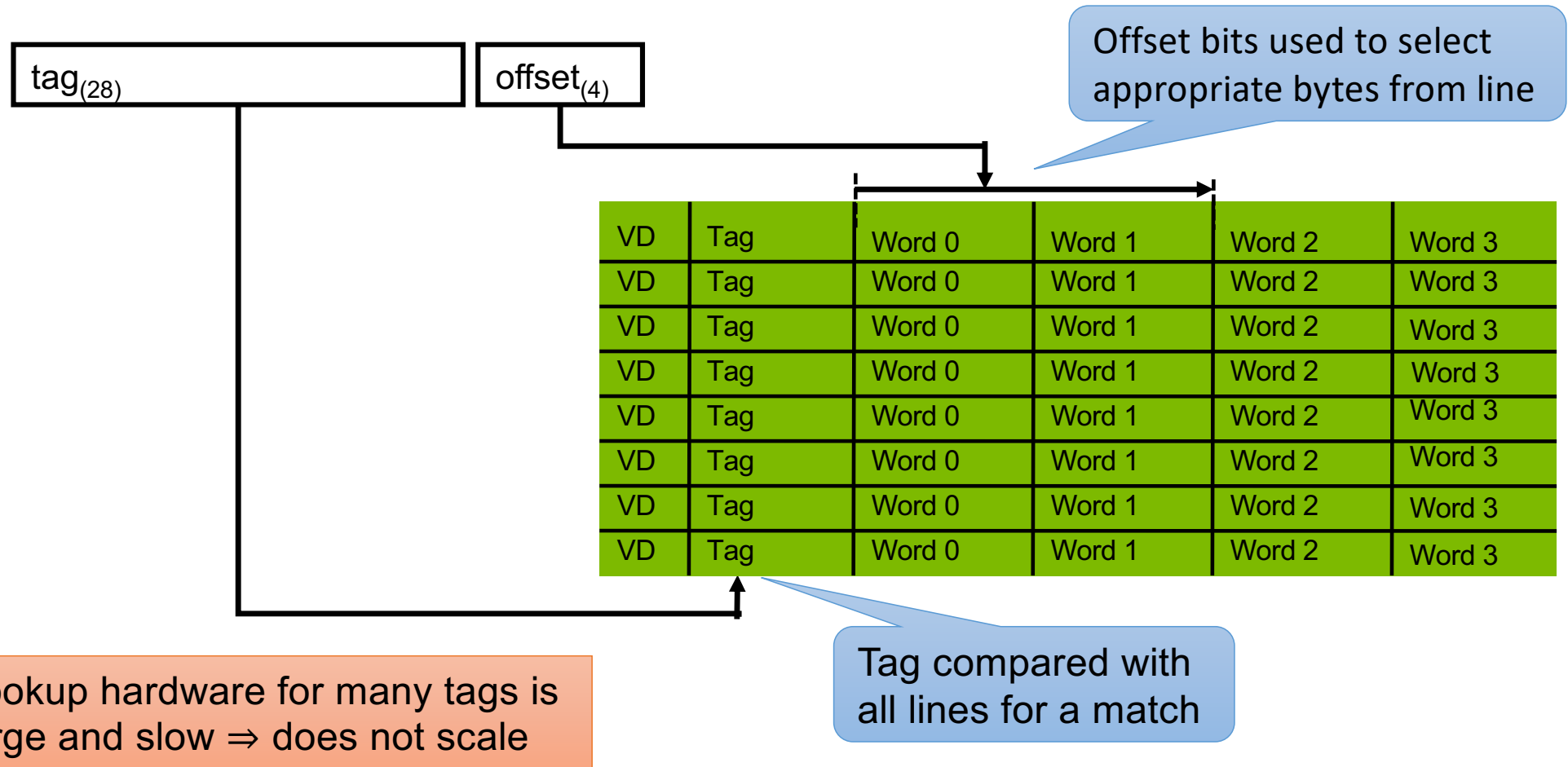
Cache Indexing: Direct Mapped



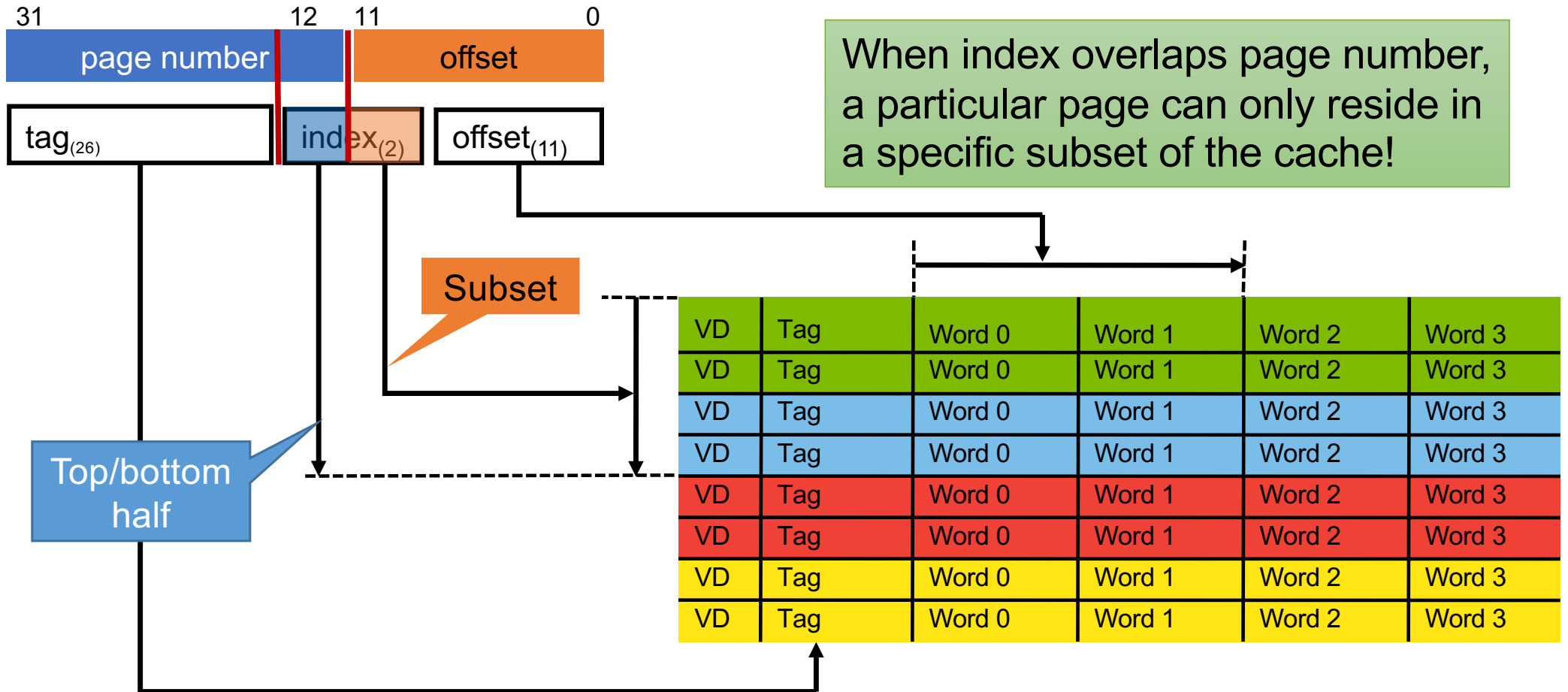
Cache Indexing: 2-Way Associative



Cache Indexing: Fully Associative

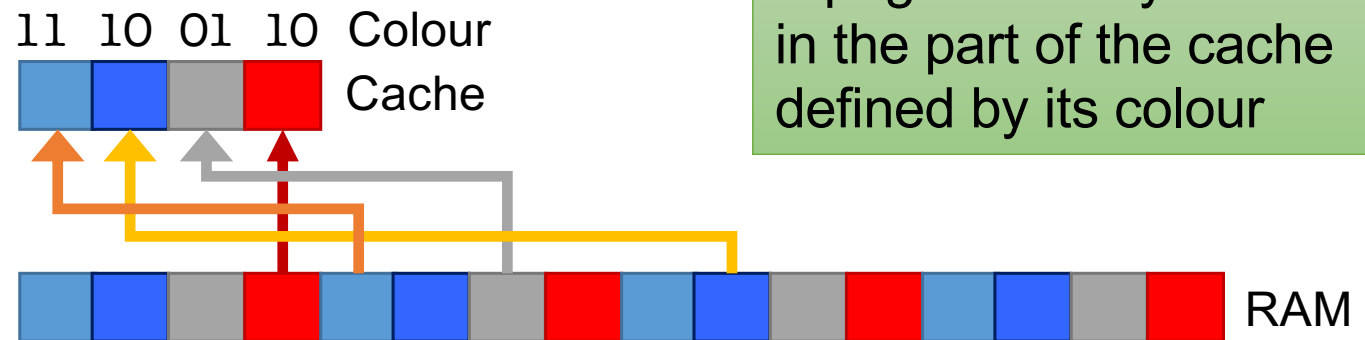


Cache Associativity vs Paging



Cache Mapping Implications

Multiple memory locations map to same cache line



A page can only reside in the part of the cache defined by its colour

If c index bits overlap page #, a page can only reside in 2^{-c} of the cache

Cache is said to have 2^c colours
 $2^c = \text{cache_size} / (\text{page_size} \times \text{assoc})$

Cache Misses

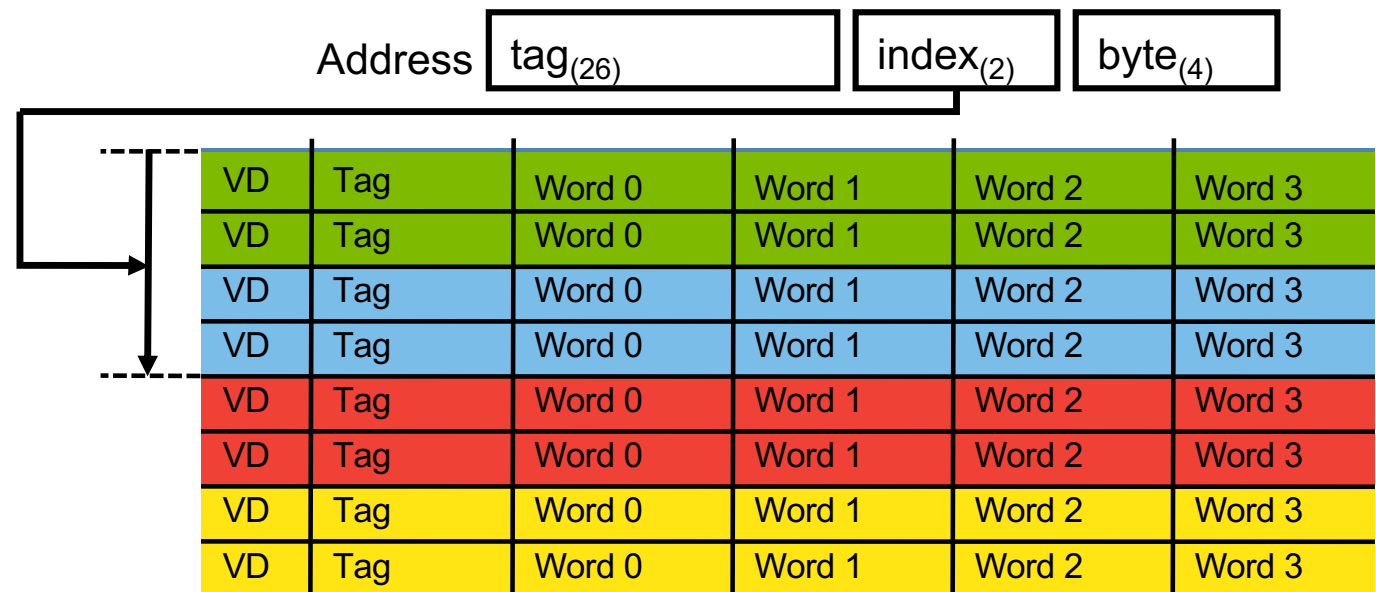
- *n-way* associative cache can hold *n lines* with the same *index* value
- More than *n lines* are competing for same index forces a miss!
- There are four different types of cache misses (“*the four Cs*”):
 - **Compulsory miss**: data cannot be in the cache (of infinite size)
 - First access (after loading data into memory or cache flush)
 - **Capacity miss**: all cache entries are in use by other data
 - Would not miss on infinite-size cache
 - **Conflict miss**: all lines *with the same index value* are in use by other data
 - Would not miss on fully-associative cache
 - **Coherence miss**: miss forced by hardware coherence protocol
 - Covered later (multiprocessing lecture)

Cache Replacement Policy

- Indexing (using address) points to specific line set
- On miss (no match and all lines valid): *replace* existing line
 - Dirty-bit determines whether write-back needed
- Replacement strategy must be simple (hardware!)

Typical policies:

- LRU
- 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
 - ✘ multi-processor cache-coherency challenge
 - **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

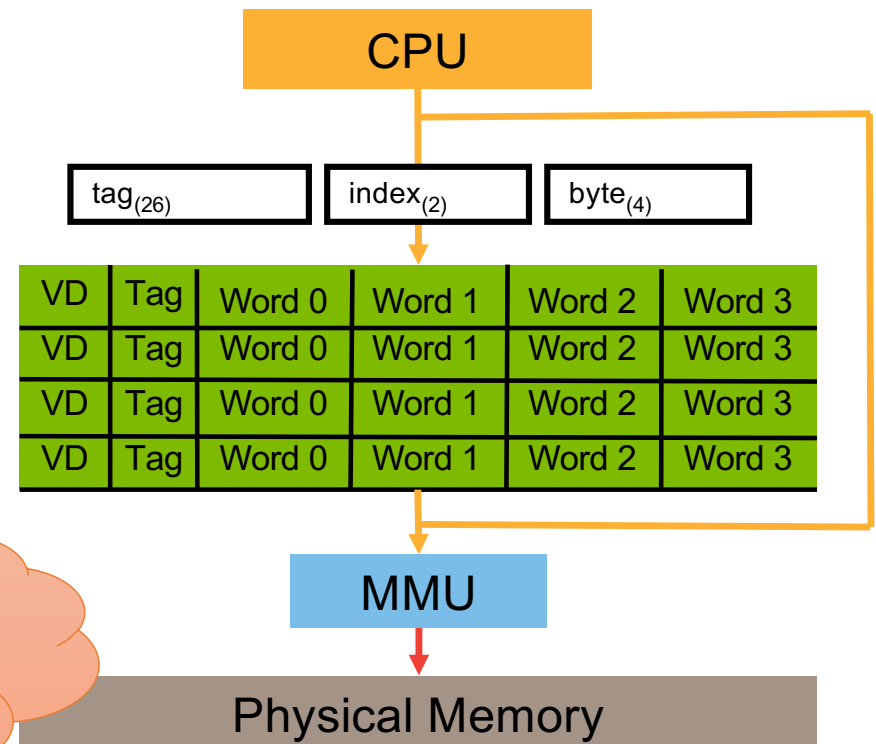
- So far pretended cache only sees one type of address: virtual or physical
- However, *indexing and tagging can use different addresses!*
- Four possible addressing schemes:
 - *virtually-indexed, virtually-tagged* (VV) cache
 - *virtually-indexed, physically-tagged* (VP) cache
 - *physically-indexed, virtually-tagged* (PV) cache
 - *physically-indexed, physically-tagged* (PP) cache

Nonsensical except with weird MMU designs

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
- Usable for on-core L1
 - Rarely used these days

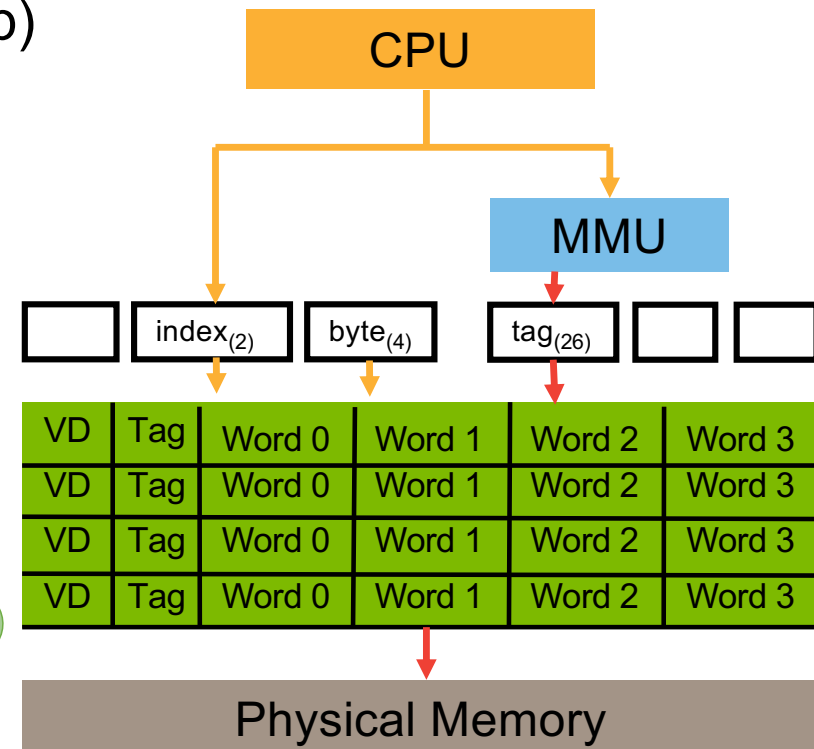
Permissions?
Write back?



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
- Used for on-core L1

Use MMU for tag check & permissions



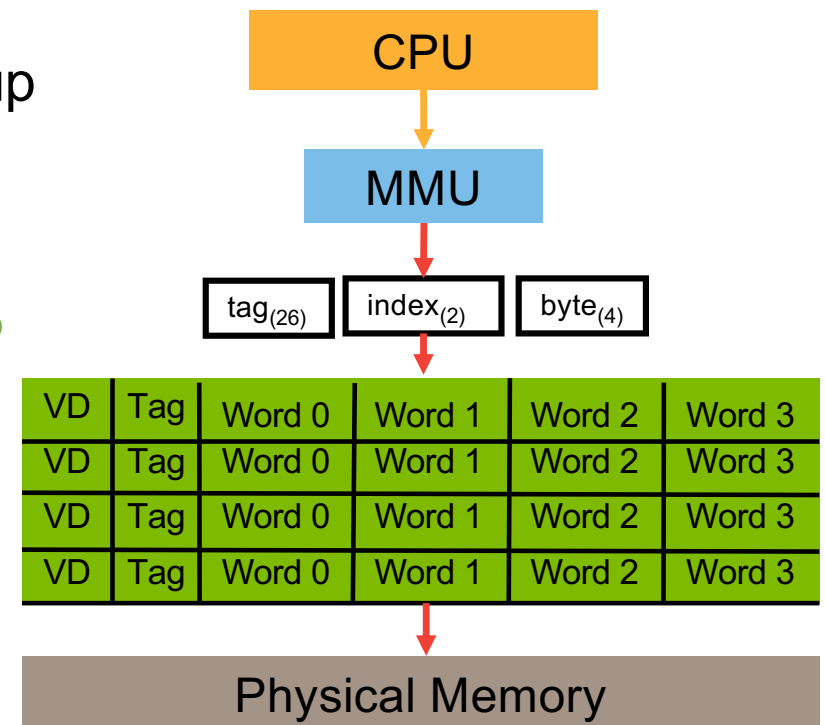
Physically-Indexed, Physically-Tagged Cache

- Only uses physical addresses
- Address translation result needed for lookup
- Only sensible choice for L2...LLC

Speed matters
less after L1 miss

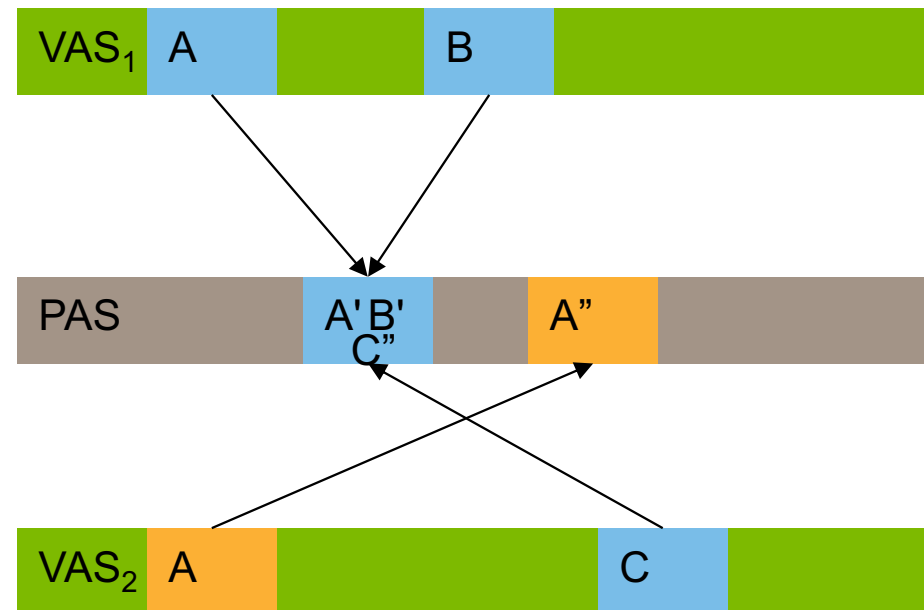
Page offset invariant under VA→PA:

- Index bits \subset offset bits
⇒ don't need MMU for indexing!
- VP = PP in this case
⇒ fast, suitable for L1
- **Single-colour cache!**



Cache Issues

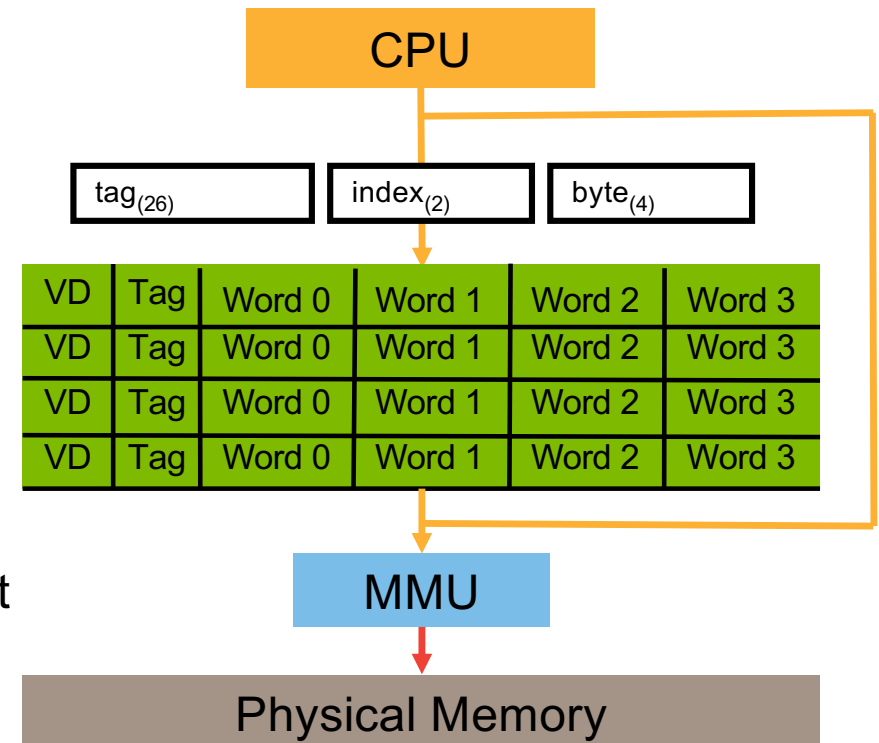
- Caches are managed by hardware transparently to software, so OS doesn't have to worry about them, ~~right?~~ **Wrong!**
- Software-visible cache effects:
 - *performance*
 - cache-friendly data layout
 - *homonyms*:
 - same address, different data
 - can affect correctness!
 - *synonyms (aliases)*: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



Virtually-Indexed Cache Issues

Homonyms – same name for different data:

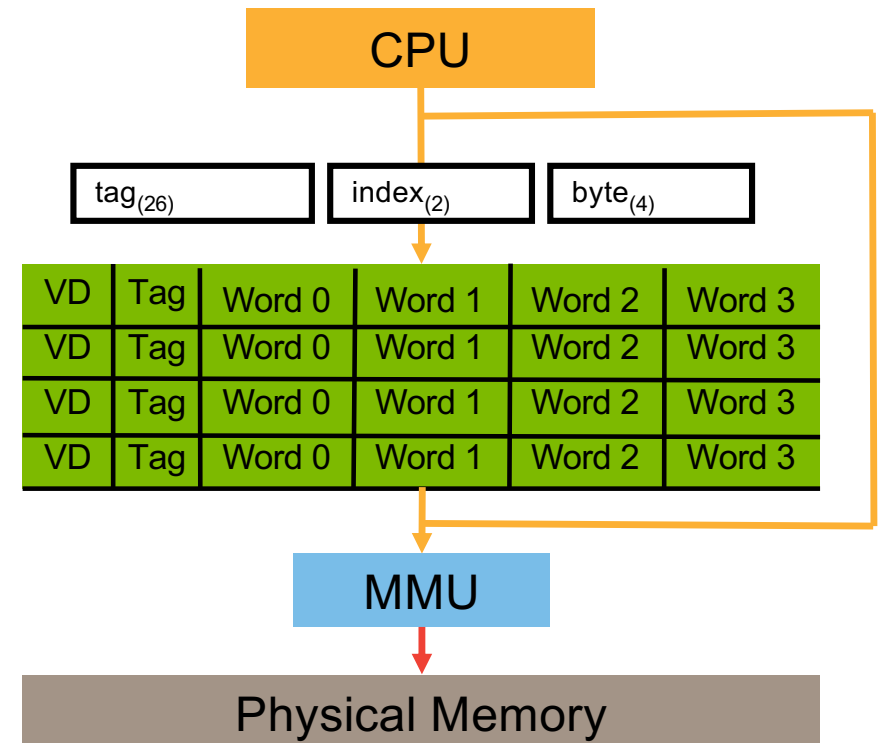
- Problem: VA used for indexing is context-dependent
 - same VA refers to different PAs
 - tag does not uniquely identify data!
 - wrong data may be accessed
 - an issue for most OSes
- Homonym prevention:
 - flush cache on each context switch
 - force non-overlapping address-space layout
 - single-address-space 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
 - ... if aliases differ in colour
 - on write, one synonym updated
 - read on other synonym returns old value
 - physical tags or ASIDs don't help!
- Are synonyms a problem?
 - depends on page and cache size (colours)
 - no problem for R/O data or I-caches

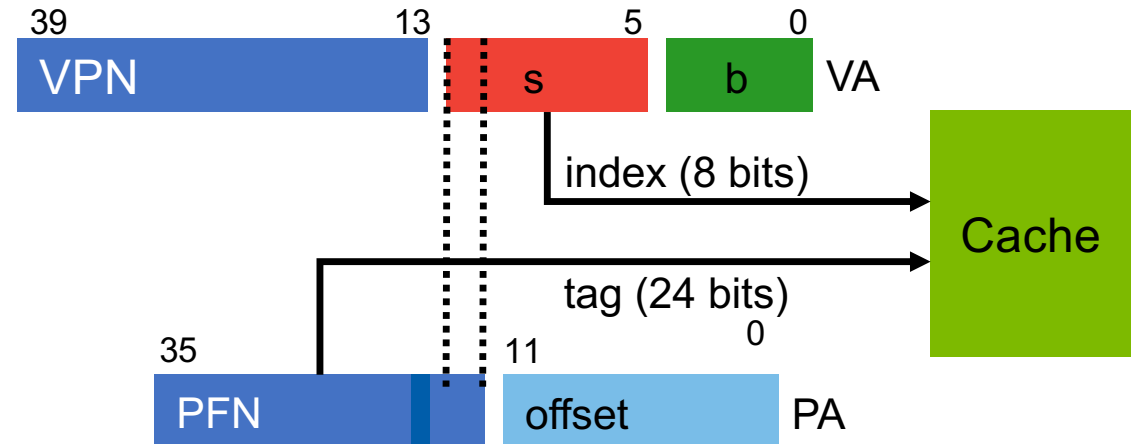


Example: MIPS R4x00 Synonyms

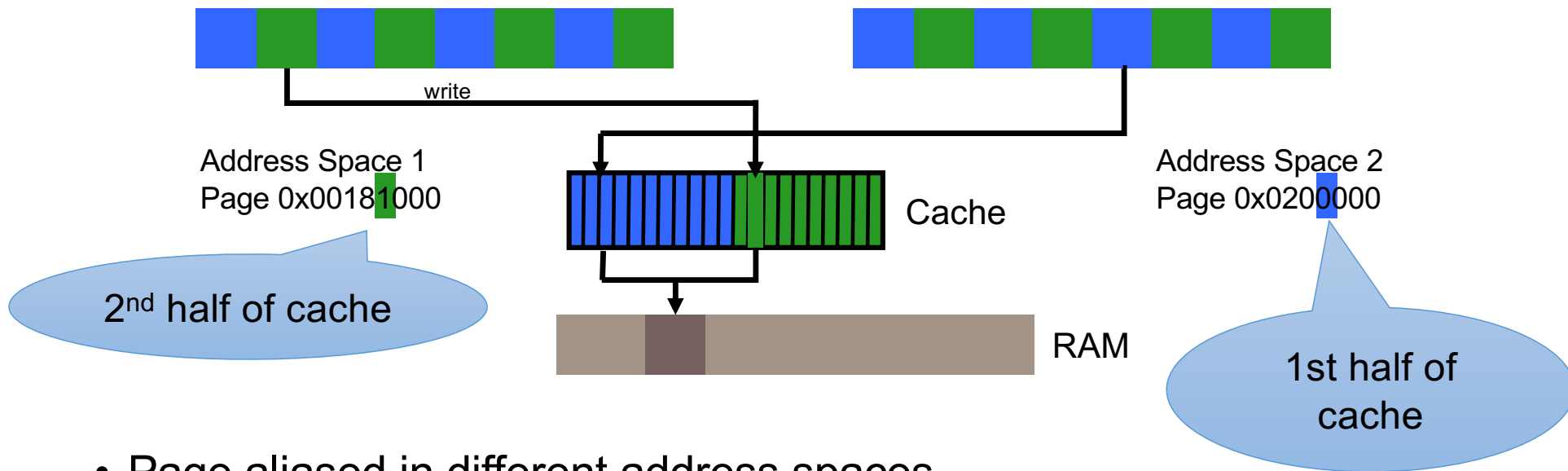
- ASID-tagged, on-chip VP cache
 - 16 KiB cache, 2-way set associative, 32 B line size, 4 KiB (base) page size
 - size/associativity = 16/2 KiB = 8 KiB > page size (2 page colours)
 - 16 KiB / (32 B/line) = 512 lines = 256 sets \Rightarrow 8 index bits (12..5)
 - overlap of tag bits and index bits, but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

Remember, only index determines location of data!

- Tag only confirms hit
- Synonym problem iff $VA_{12} \neq VA'_{12}$
- Problem of virtually-indexed cache with multiple colours

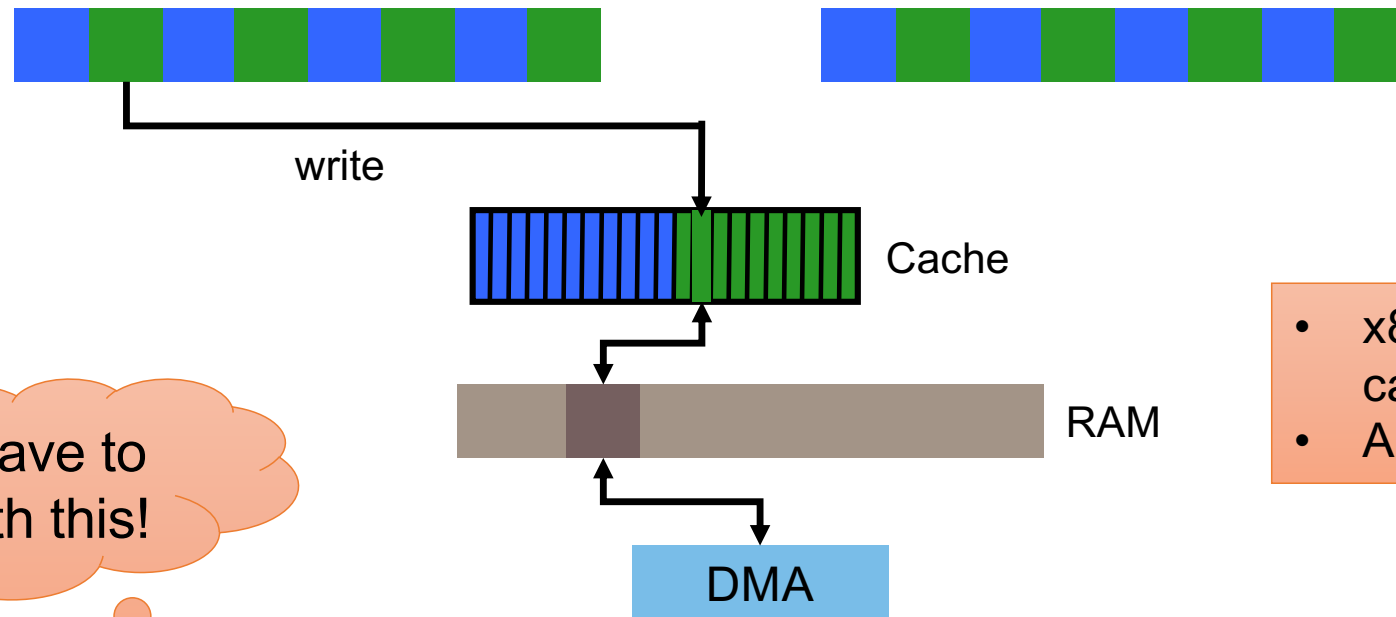


Address Mismatch Problem: Aliasing



- 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*

DMA Consistency Problem

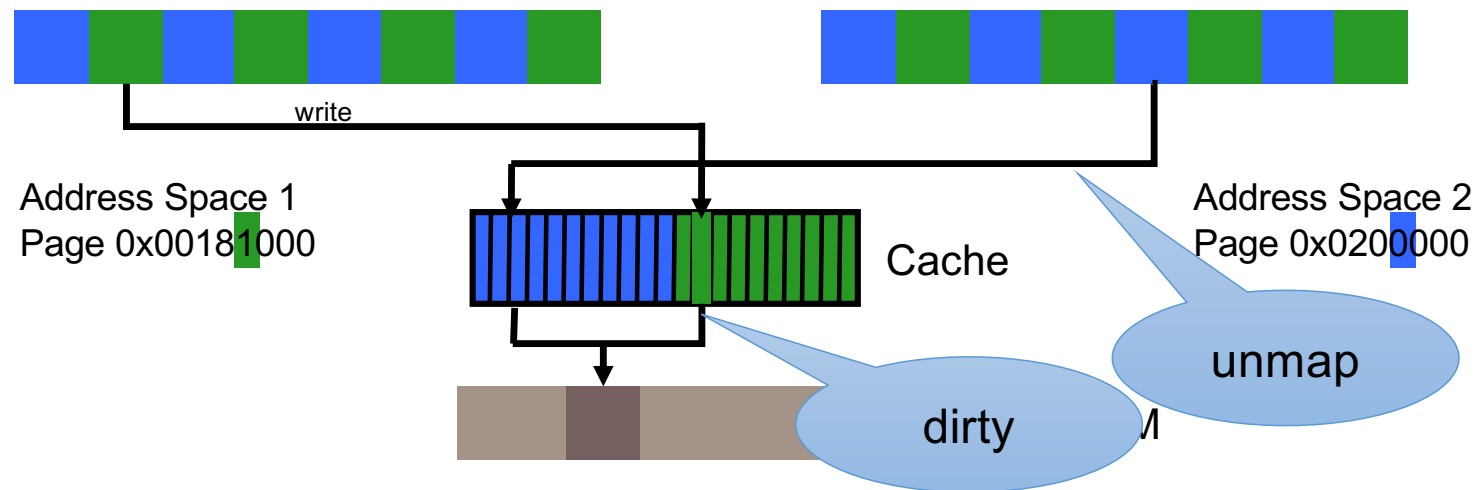


You'll have to deal with this!

- x86 ensures DMA & cache consistent
- Arm does not!

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

Address Mismatch Problem: Aliasing

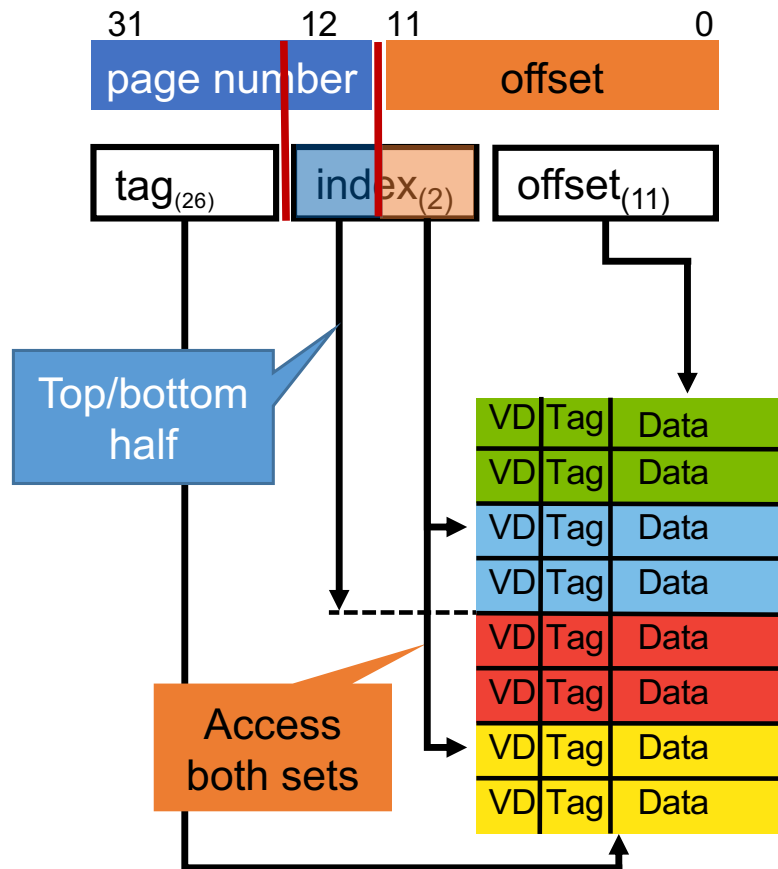


- Unmap aliased page, remaining page has a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Access new page
 - without replication, new write will overwrite old (hits same cache line)
 - with replication, alias may write back after remapping: “*cache bomb*”

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 $VA_{12} = PA_{12}$ – colour memory!
- Hardware synonym detection

Hardware Synonym Detection (Arm A53)



- Lookup accesses sets of both colours
- If tag matches in both set: have a synonym
- If the access is a store then invalidate the synonym of the “wrong” colour
- VP cache behaves like PP despite multiple colours

Summary: VV Caches

- ✓ Fastest (don't rely on TLB for retrieving data)
 - ⌘ still need TLB lookup for protection
 - ⌘ ... or alternative mechanism for providing protection
 - ⌘ still need TLB lookup or physical tag for writeback
- ⌘ 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
- Used on MC68040, i860, ARM7/ARM9/StrongARM/Xscale
- Used for I-caches on several other architectures (Alpha, Pentium 4)
- Not used on recent architectures

Historically used with shallow hierarchies to support bigger L1

Summary: ASID-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
- Not used on recent architectures

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 contemporary architectures for L1 cache
 - but mostly single-colour (pseudo-PP) or with HW alias prevention (Arm)

Summary: PP Caches

⌘ Slowest

⌘ requires result of address translation before lookup starts

✓ No synonym problem

✓ No homonym problem

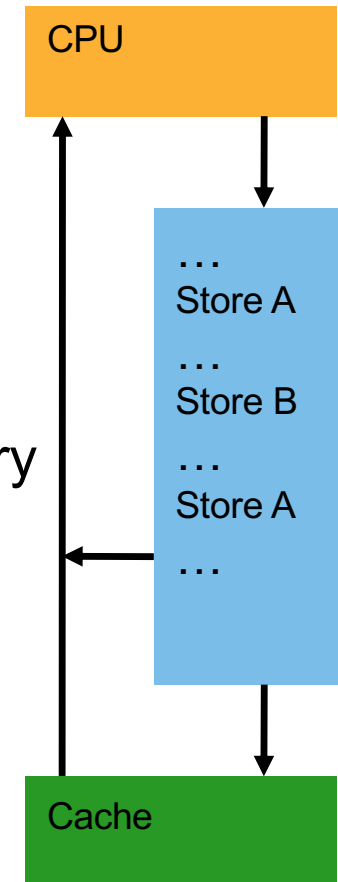
✓ Easy to manage

✓ Cache can use *bus snooping* for DMA/multicore coherency

✓ Obvious choice for L2–LLC where speed matters less

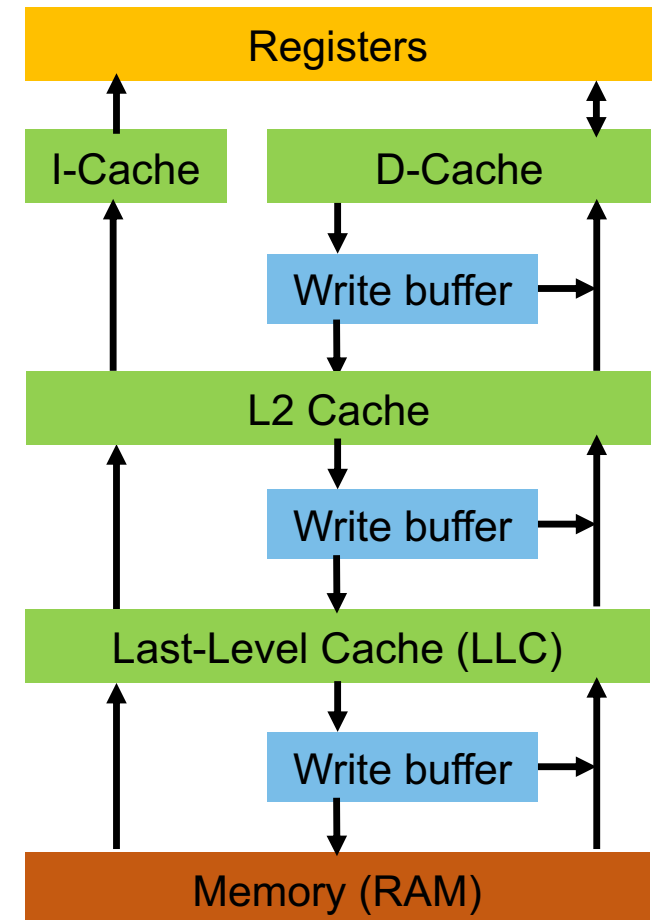
Write Buffer

- 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*
 - May exist at any cache level, or between cache and memory
- Can fetch intermediate values out of buffer
 - to service read 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 memory ordering*”, to be revisited in SMP context

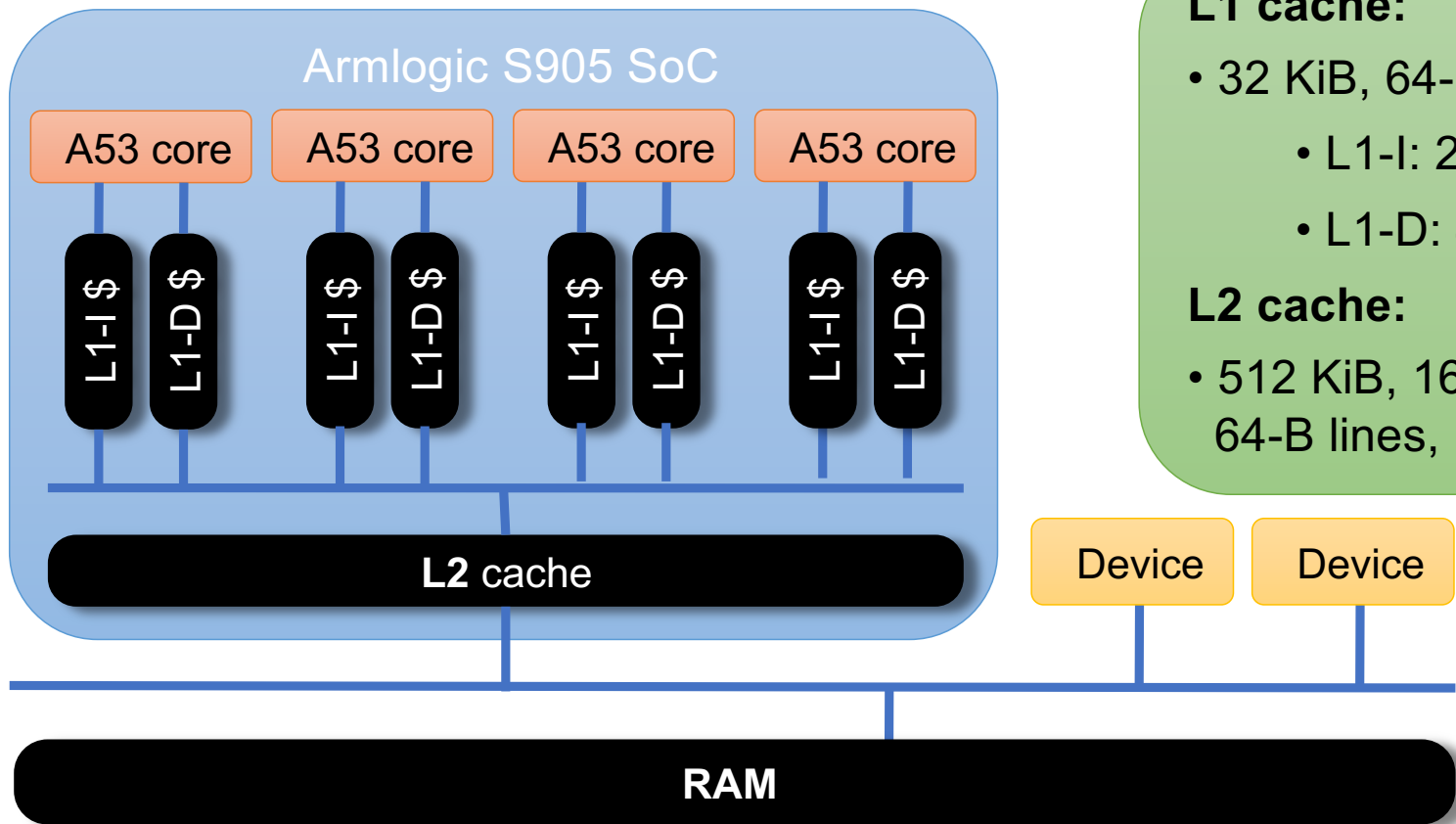


Cache Hierarchy

- Hierarchy of caches to balance memory accesses:
 - small, fast, virtually-indexed L1
 - large, slow, physically indexed L2–LLC
- Each level reduces and clusters traffic
- L1 split into I- and D-caches
 - “Harvard architecture”
 - requirement of pipelining
- Other levels unified
- Chip multiprocessors (aka multicores):
 - Usually LLC shared chip-wide
 - L2 private (Intel) or clustered (AMD)



ODROID-C2 (Cortex A53) System Architecture



L1 cache:

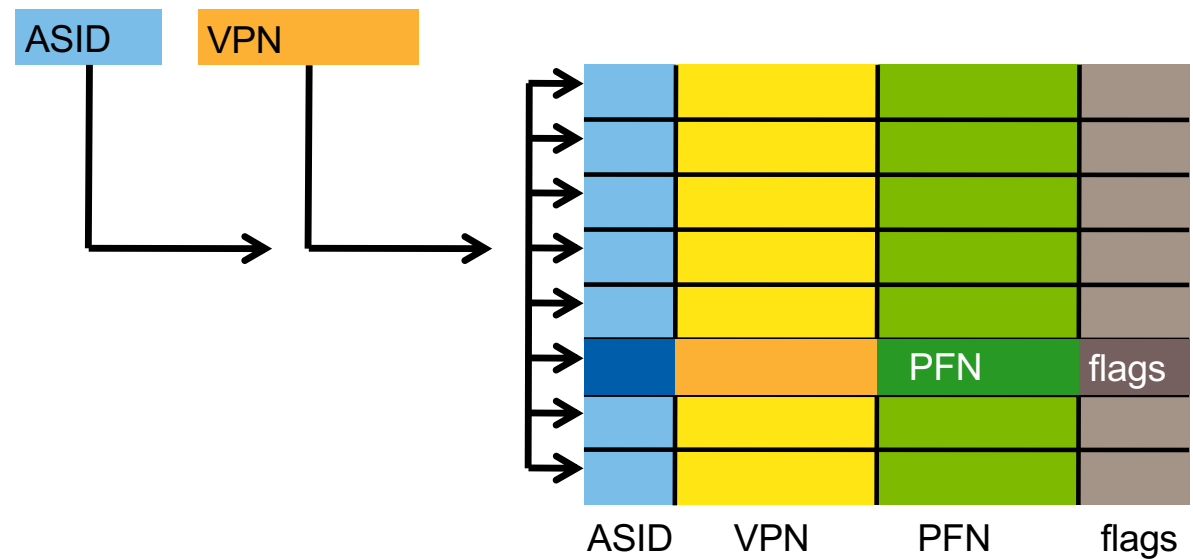
- 32 KiB, 64-B lines
 - L1-I: 2-way, virtually addr.
 - L1-D: 4-way, “physically” addr.

L2 cache:

- 512 KiB, 16-way
64-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



TLB Size (I-TLB+D-TLB)

Not much growth in 40 years!

Architecture	Size (I+D)	Assoc	Page Size	Coverage
VAX-11	64–256	2	0.5 KiB	32–128 KiB
ix86	32i + 64d	4	4 KiB + 4 MiB	128 KiB
MIPS	96–128	full	4 KiB – 16 MiB	384–512 KiB
SPARC	64	full	8 KiB – 4 MiB	512 KiB
Alpha	32–128i + 128d	full	8 KiB – 4 MiB	256 KiB
RS/6000 (PPC)	32i + 128d	2	4 KiB	256 KiB
Power-4 (G5)	1024	4	4 KiB	512 KiB
PA-8000	96i + 96d	full	4 KiB – 64 MiB	384 KiB
Itanium	64i + 96d	full	4 KiB – 4 GiB	384 KiB
ARMv7 (A9)	64–128	1–2	4 KiB – 16 MiB	256–512 KiB
x86 (Skylake)	L1:128i+64d; L2:1536	4	4 KiB + 2/4 MiB	1 MiB

TLB Size

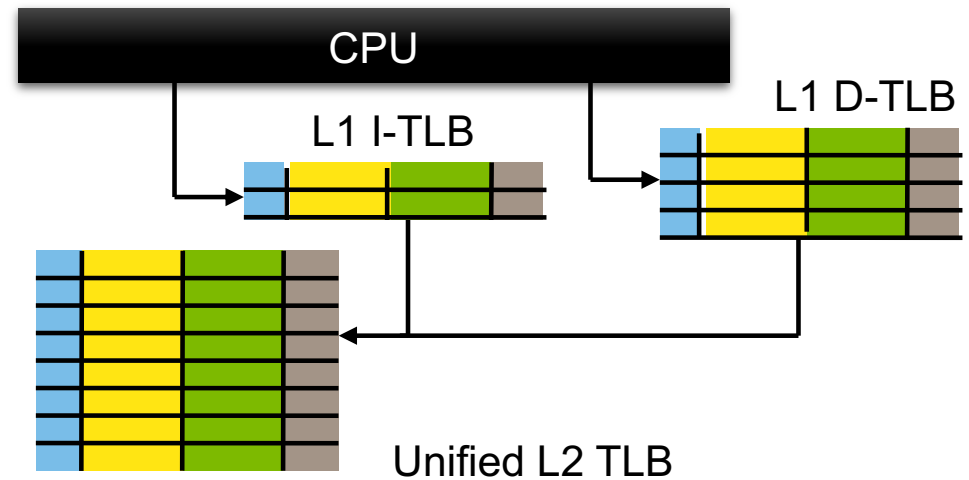
TLB coverage

- Memory sizes are increasing
- Number of TLB entries are roughly constant
- Base page sizes are steady
 - 4 KiB (SPARC, Alpha used 8KiB)
 - OS designers have trouble using superpages effectively
- 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!

TLB can become a bottleneck!

Multi-Level TLBs

- Multi-level design (like I/D cache)
- Improve size-performance tradeoff



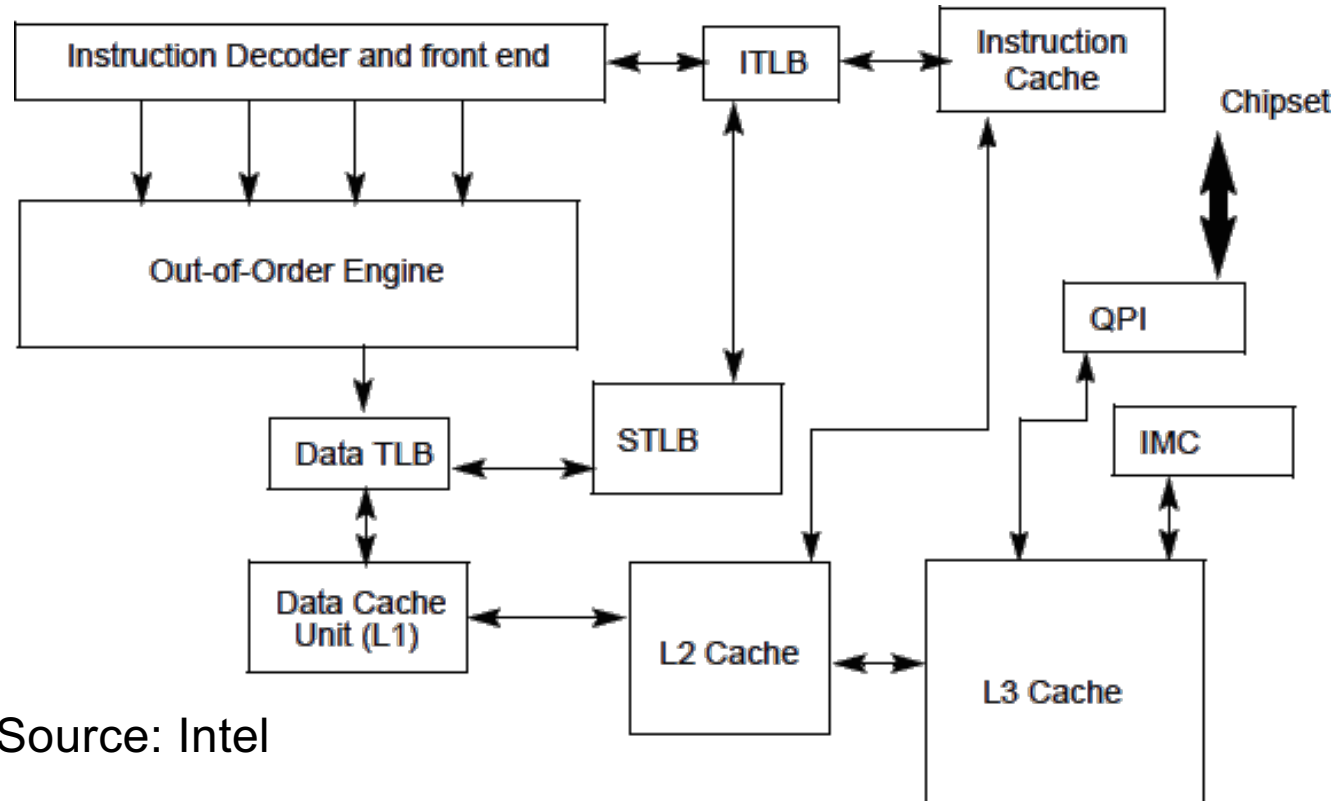
Intel Core i7

L	I/D	Pages	Assoc	Entr
1	I	4 KiB	4-way	64
1	D	4 KiB	4-way	64
1	I	2/4 MiB	fully	7
1	D	2/4 MiB	4-way	32
2	unif	4 KiB	4-way	512

Arm A53

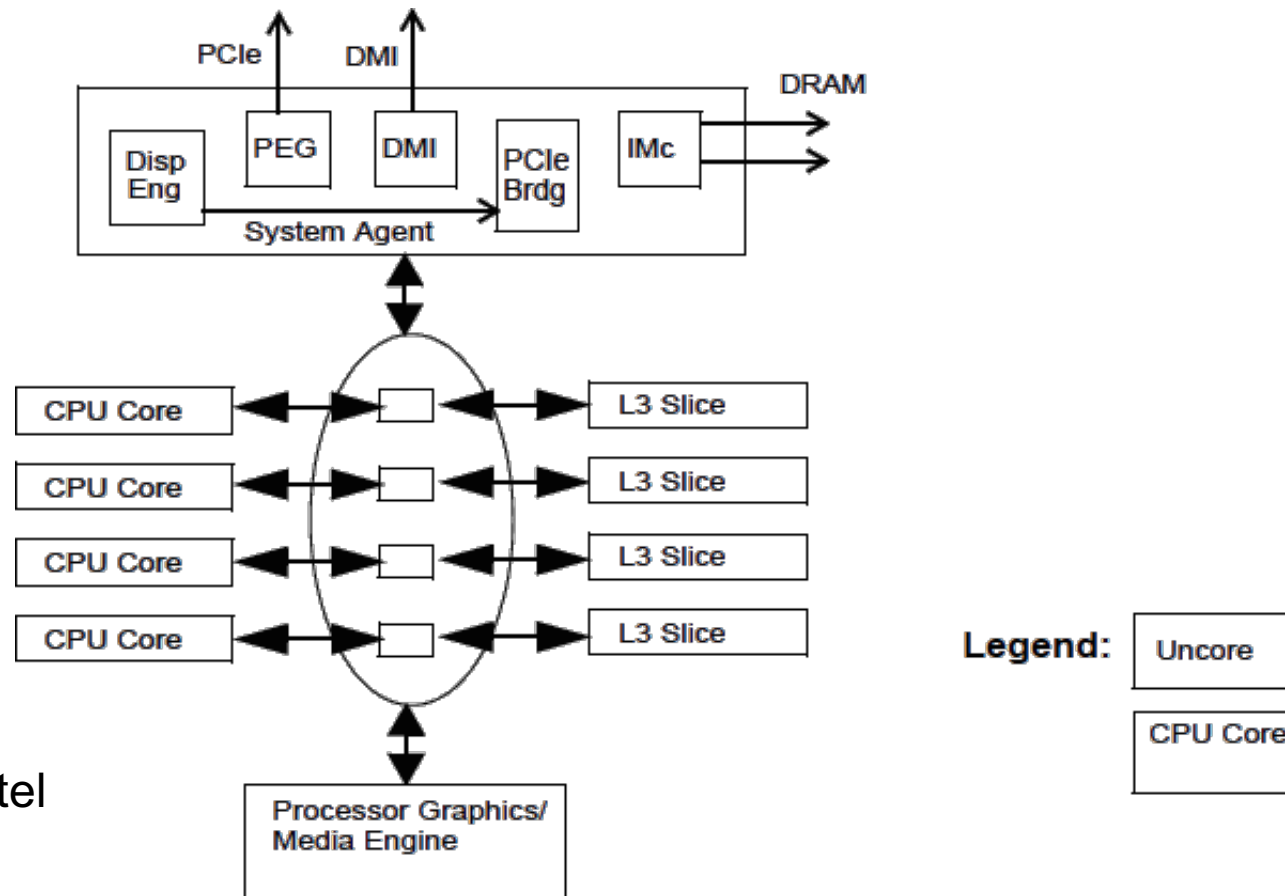
L	I/D	Pages	Assoc	#
1	I	4 KiB–1 GiB?	full?	10
1	D	4 KiB–1 GiB?	full?	10
2	unif	4 KiB–512 MiB	4-way	512

Intel Core i7 (Haswell) Cache Structure



Source: Intel

Intel Haswell L3 Cache



Source: Intel