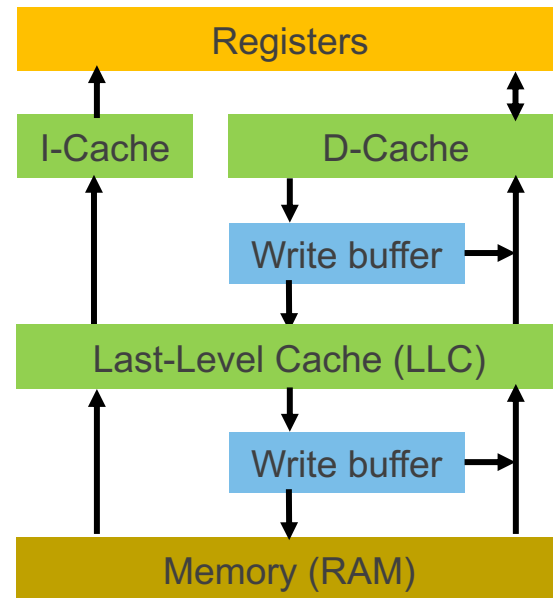


2026 T3 Week 02 Part 2

Hardware Considerations: What Every OS Designer Must Know

Gernot Heiser



Copyright Notice

**These slides are distributed under the
Creative Commons Attribution 4.0 International (CC BY 4.0) 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/4.0/legalcode>

Today's Lecture

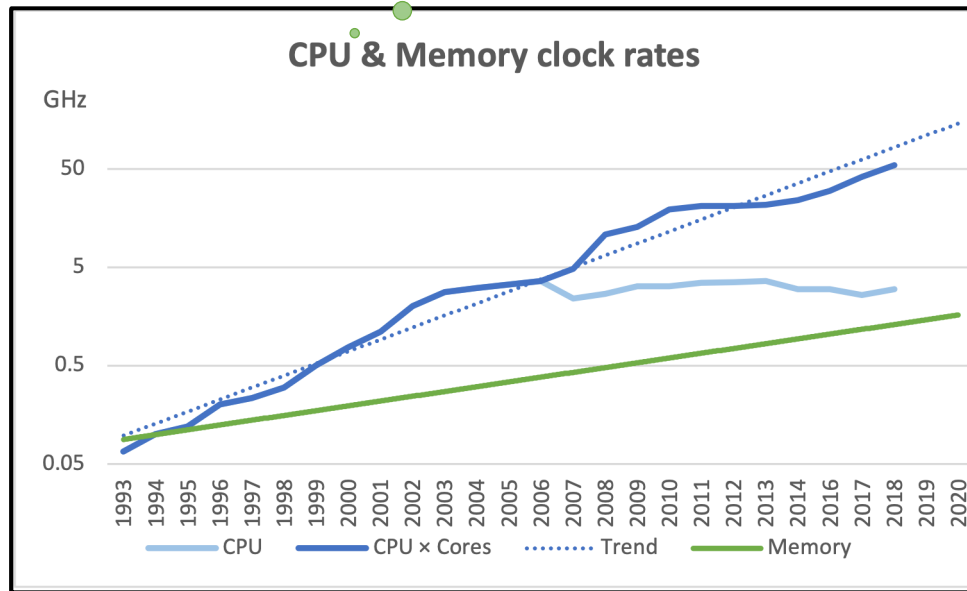
- Caches
 - What are caches, why do we have them?
 - How do they work (in detail)?
 - Why you need to understand them? – Software effects
 - Cache hierarchy
 - Translation caches: TLB
- Devices

Later: Concurrency effects and memory models

Cache Basics

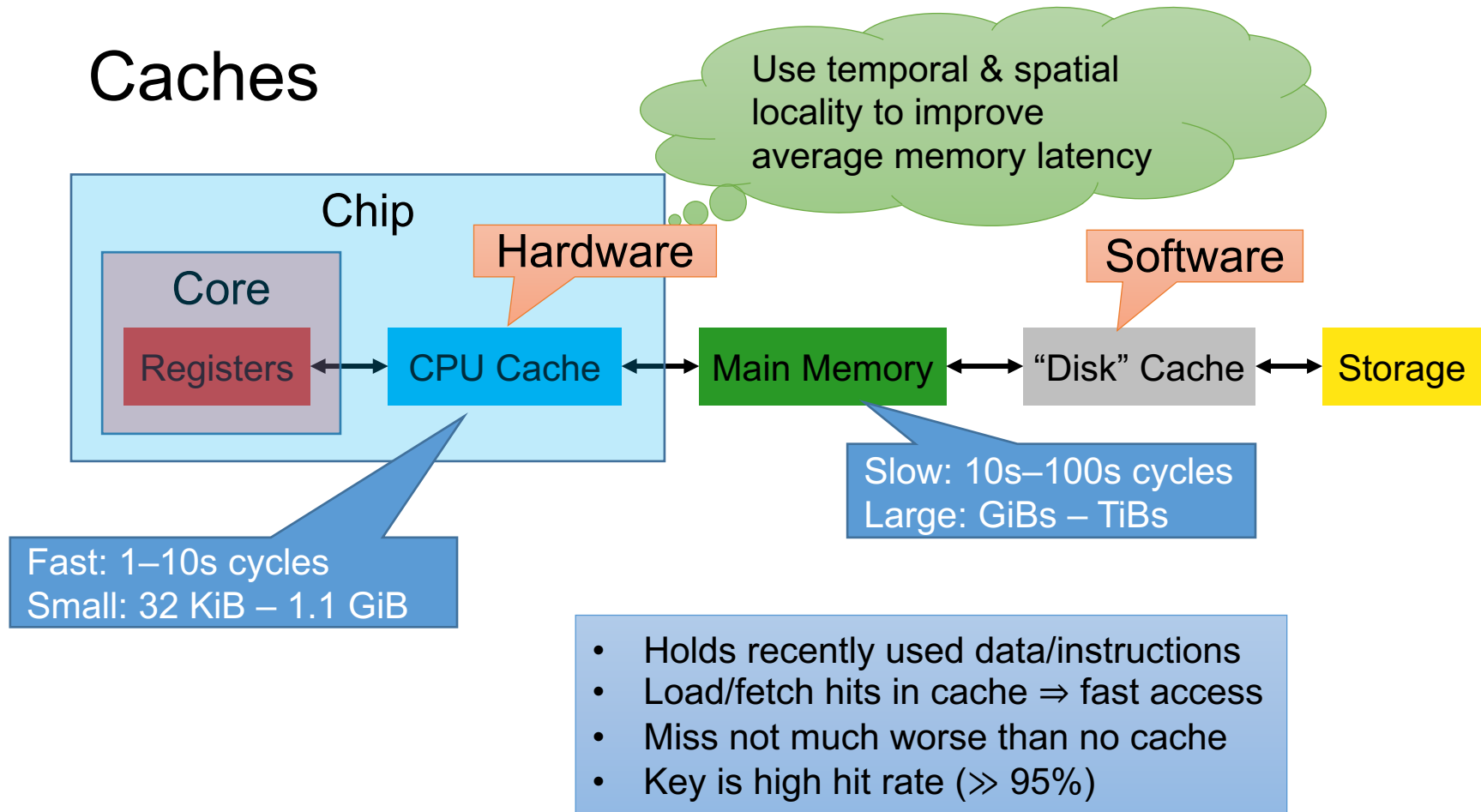
The Memory Wall

Clock rates of
Intel processors



Speed gap still
widens by approx
18% per year!

Caches



Cache Organisation: Unit of Data Transfer



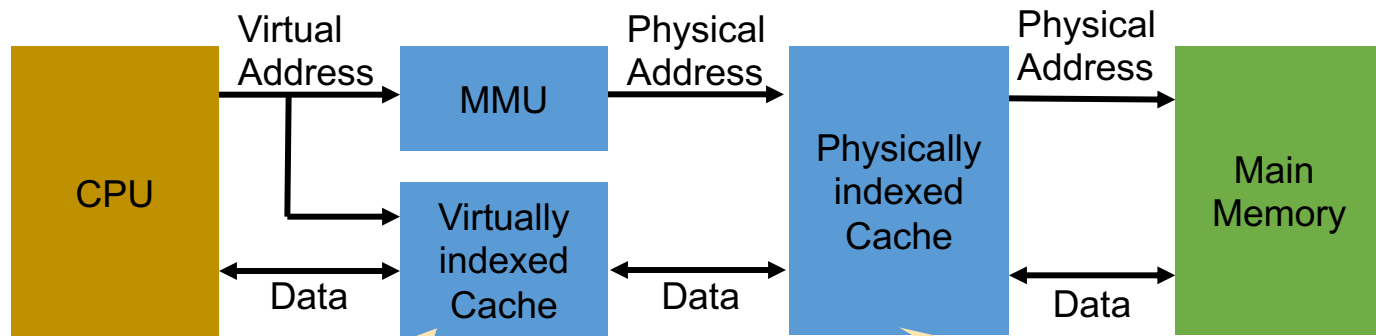
Line is also unit of allocation, holds data and

- 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

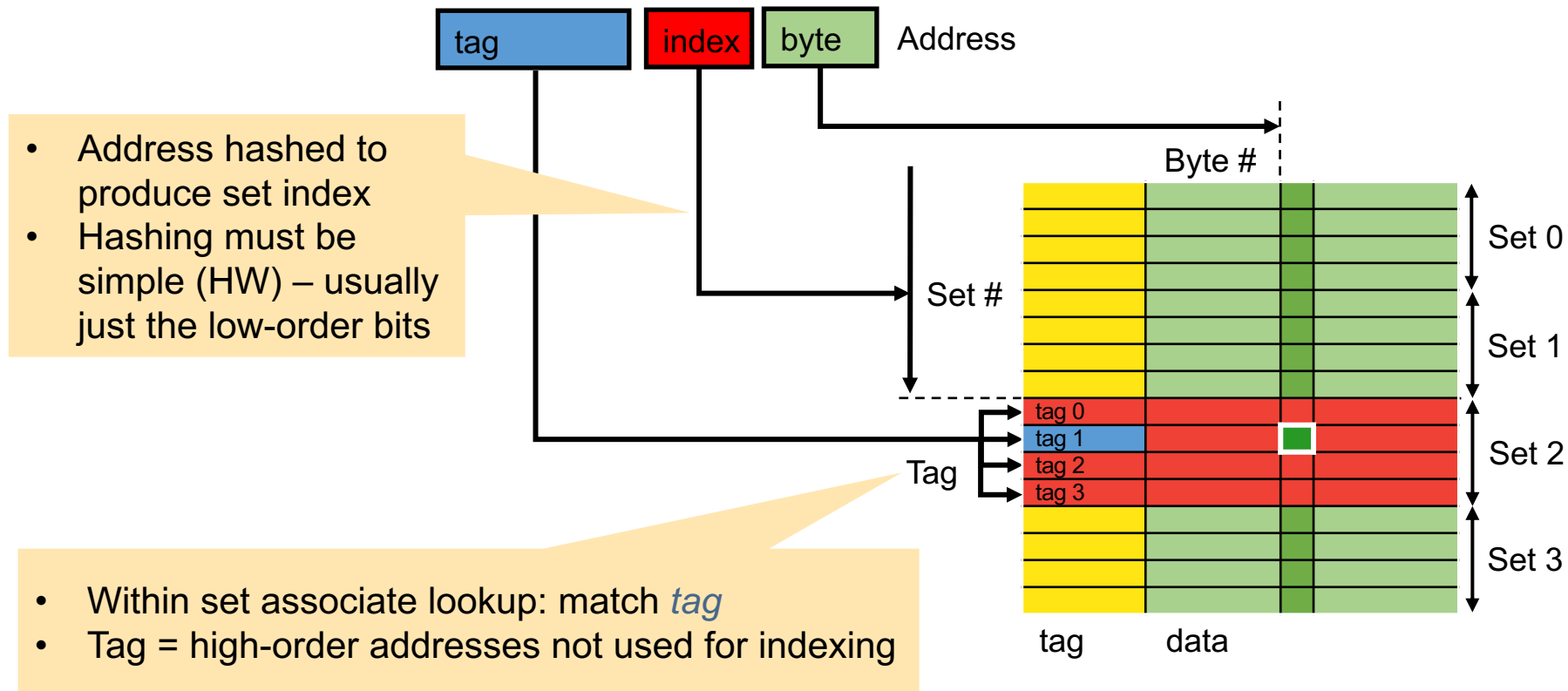
- Looked up by physical address
- Requires result of address translation

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

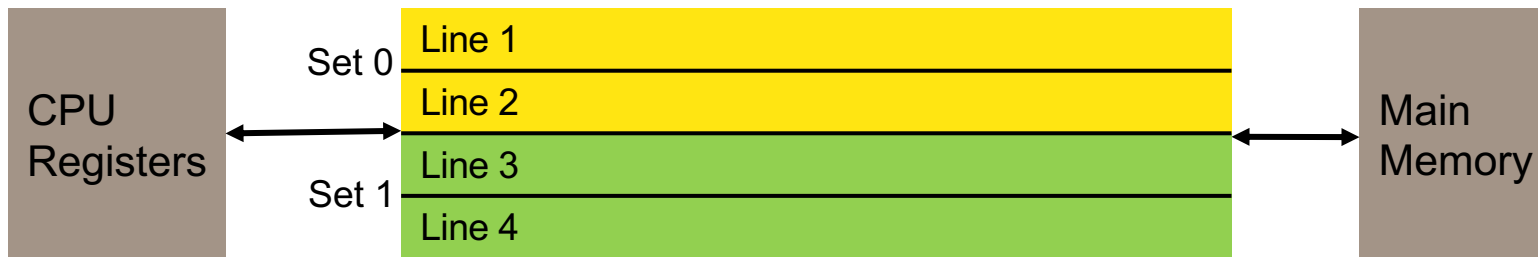
- L1 closest to CPU
- LLC: last-level cache
- Only L1 may be virtually addressed

Indexing

Cache Indexing



Cache Indexing

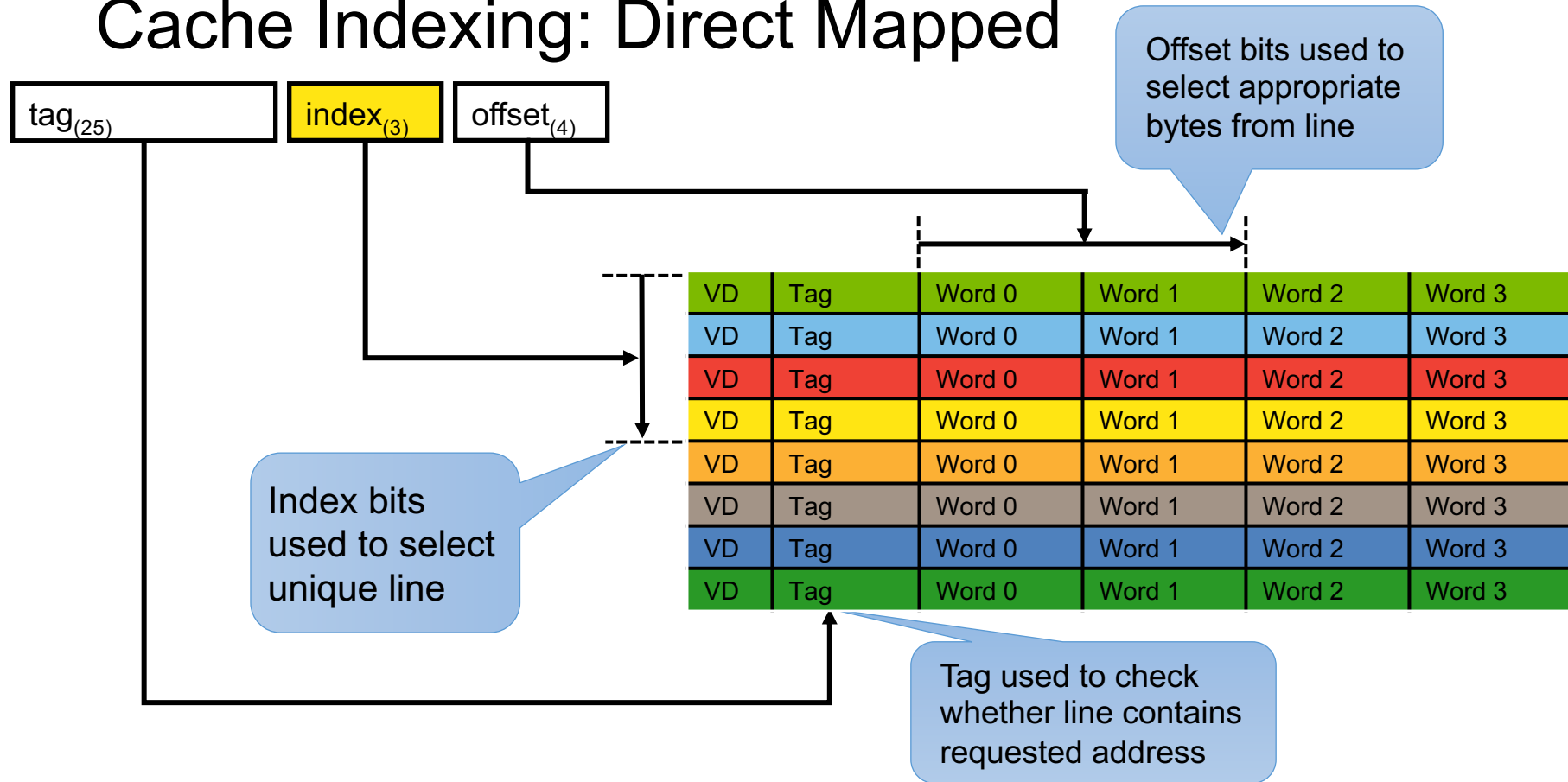


Many conflicts
⇒ low hit rate

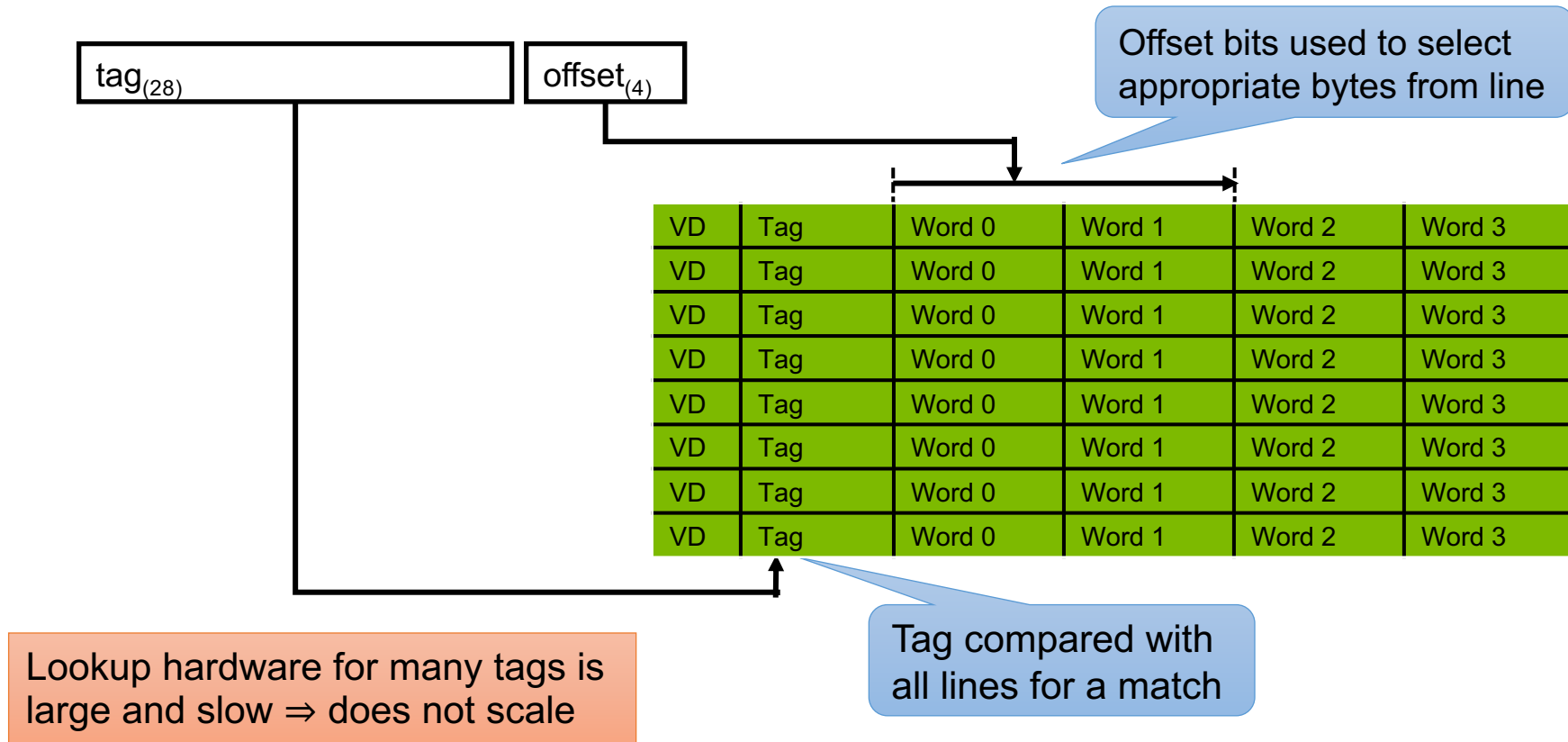
- n lines per set: n -way set-associative cache
- $n = 1$: *direct mapped*
- $2 \leq n < \# \text{ lines}$: *set associative*
- $n = \# \text{ lines}$: *fully associative*

Slow & power-hungry

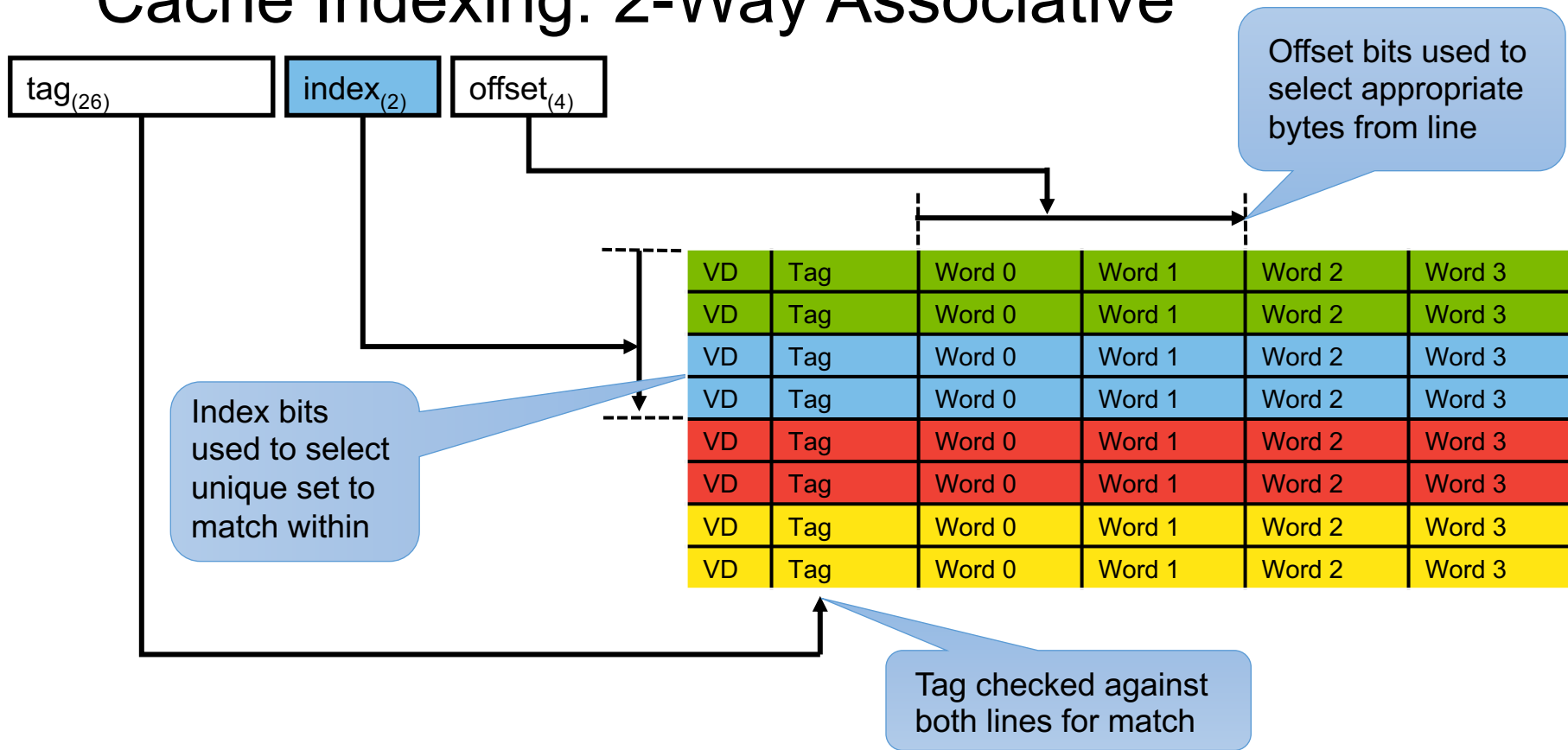
Cache Indexing: Direct Mapped



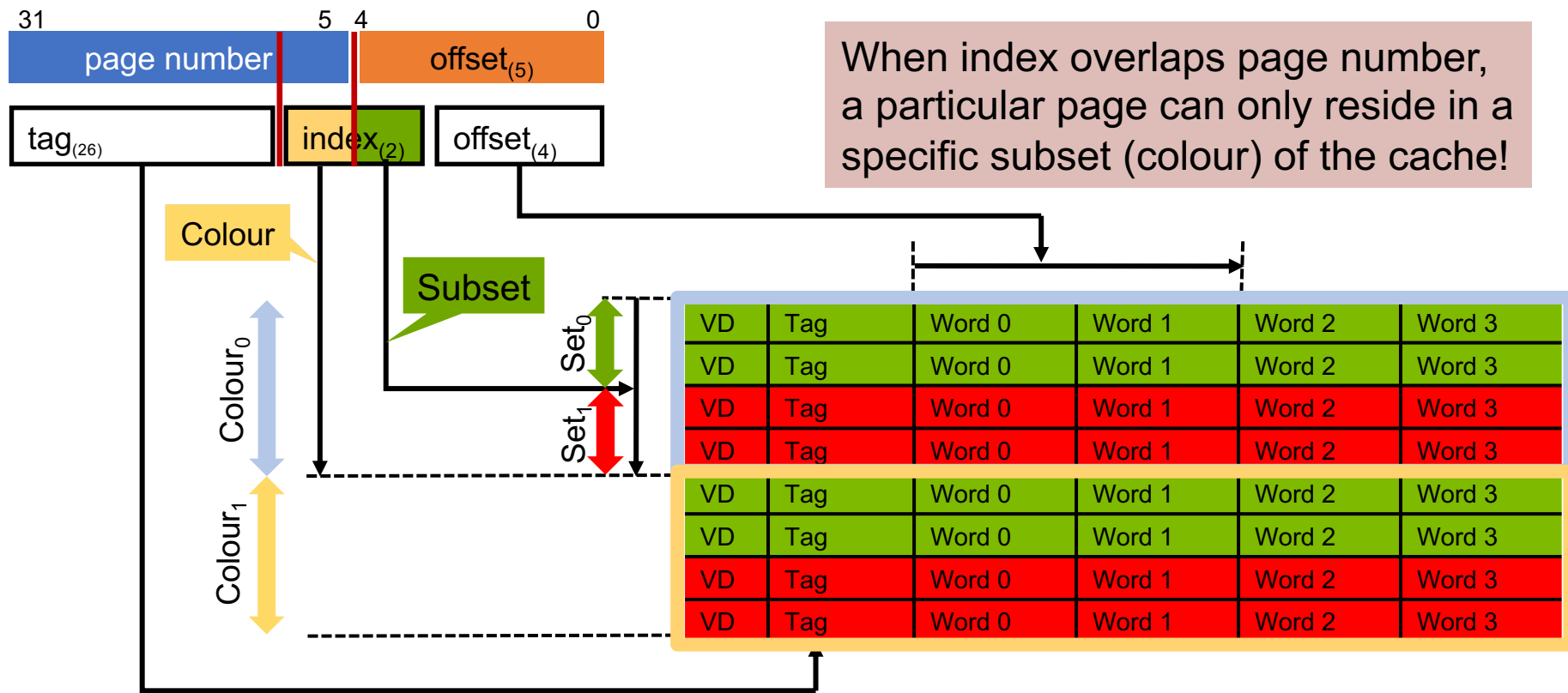
Cache Indexing: Fully Associative



Cache Indexing: 2-Way 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:
Colour = c LSB of page #

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
 $2^c = \text{cache_size} / (\text{page_size} \times \text{assoc})$

Misses & Replacement Policy

Cache Misses

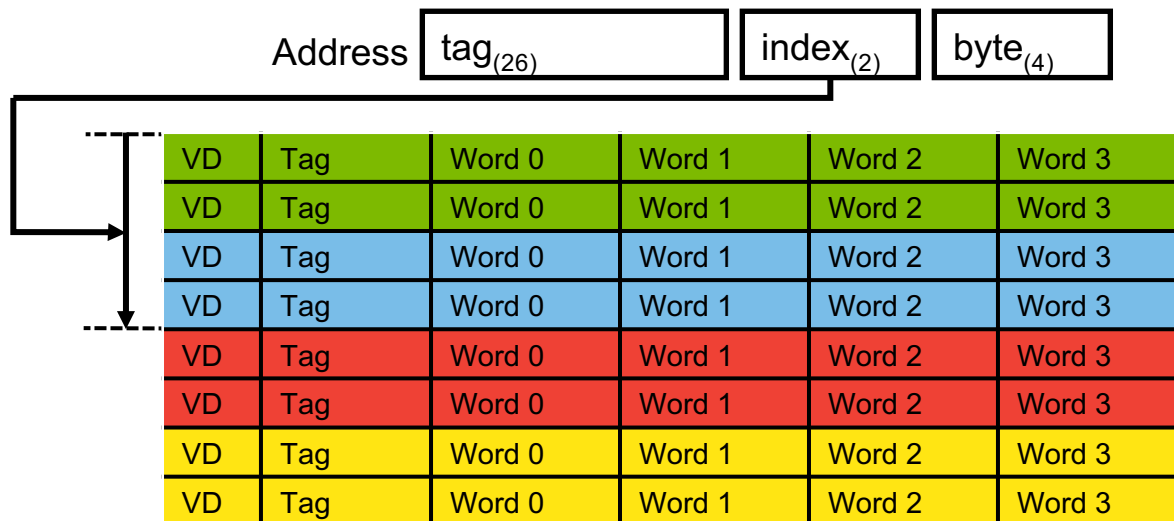
- *n-way* associative cache can hold *n lines* with the same *index* value
- More than *n lines* are competing for same index forces a miss!
- There are four different types of cache misses (“*the four Cs*”):
 - **Compulsory miss**: data cannot be in the cache (of infinite size)
 - First access (after loading data into memory or cache flush) – unavoidable
 - **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 through:** stores update cache and memory immediately
 - ✓ memory is always consistent with cache $\Rightarrow$ better data integrity and error recovery
 - ✗ higher memory/bus traffic (cache only absorbs reads)
 - **write back:** Stores only update cache; memory is updated once dirty line is replaced or explicitly flushed
 - ✓ clusters writes
 - ✗ memory inconsistent with cache
 - ✗ multi-processor cache-coherency challenge
- 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-through & no-allocate
- write-back & write allocate

Most common these days

Cache Indexing Schemes

Cache Indexing 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

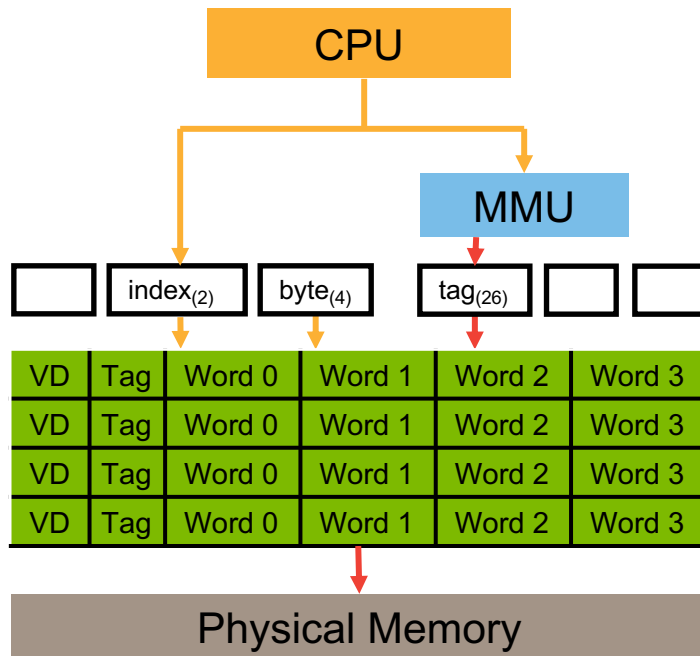
Rare these days,
will ignore

Nonsensical except with
weird MMU designs

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 access
- Used for on-core L1

Use MMU for
tag check &
permissions



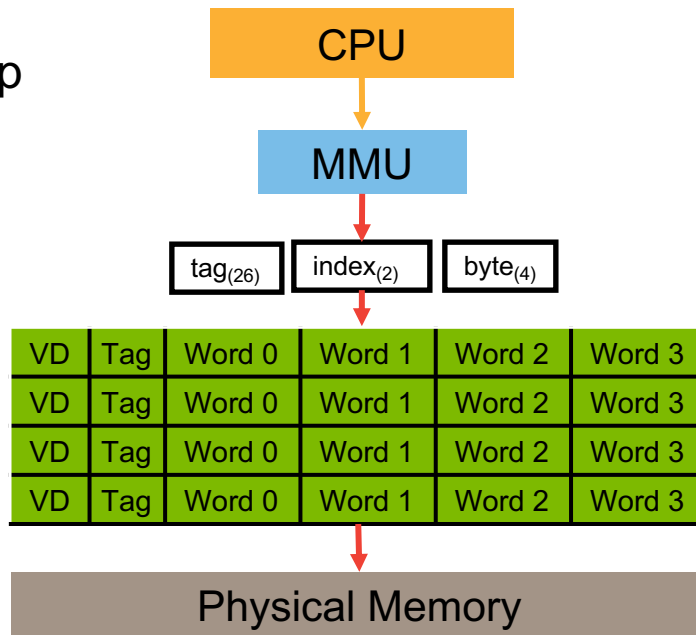
Physically-Indexed, Physically-Tagged Cache

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

After L1 miss, MMU result is available

Page offset invariant under VA→PA:

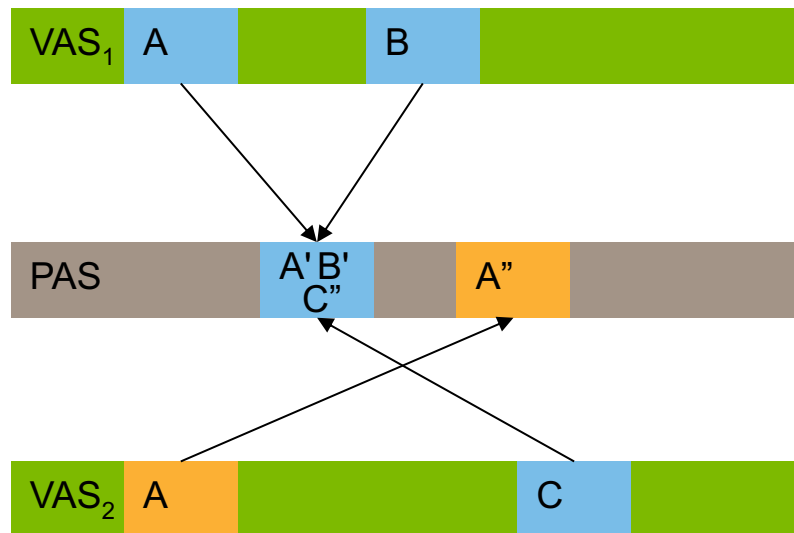
- Index bits $\subset$ offset bits
⇒ don't need MMU for indexing!
- VP = PP in this case
⇒ fast, suitable for L1
- **Single-colour cache!**



Software-Visible Effects

Cache Issues

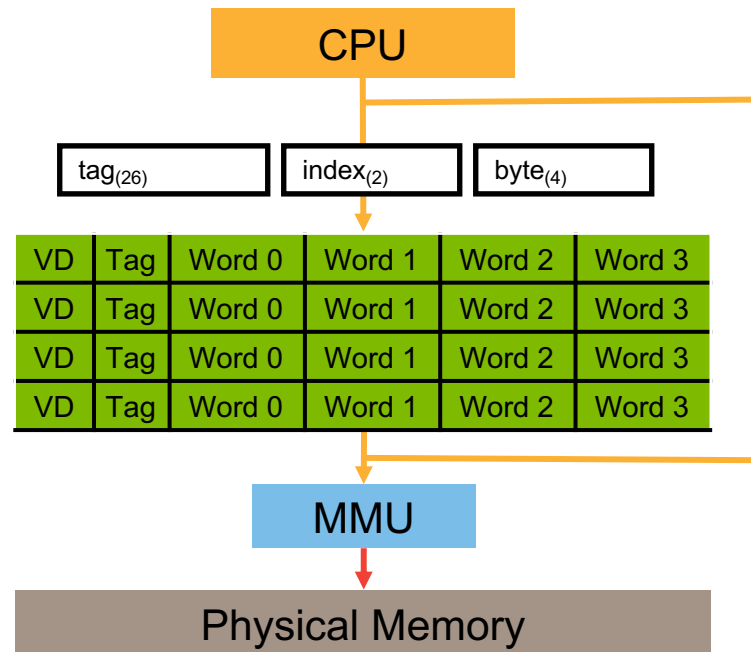
- Caches are managed by hardware transparently to software, so OS doesn't have to worry about them, ~~right?~~ **Wrong!**
- Software-visible cache effects:
 - *performance*
 - cache-friendly data layout
 - *homonyms*:
 - same address, different data
 - can affect correctness! (on VV caches – ignoring)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



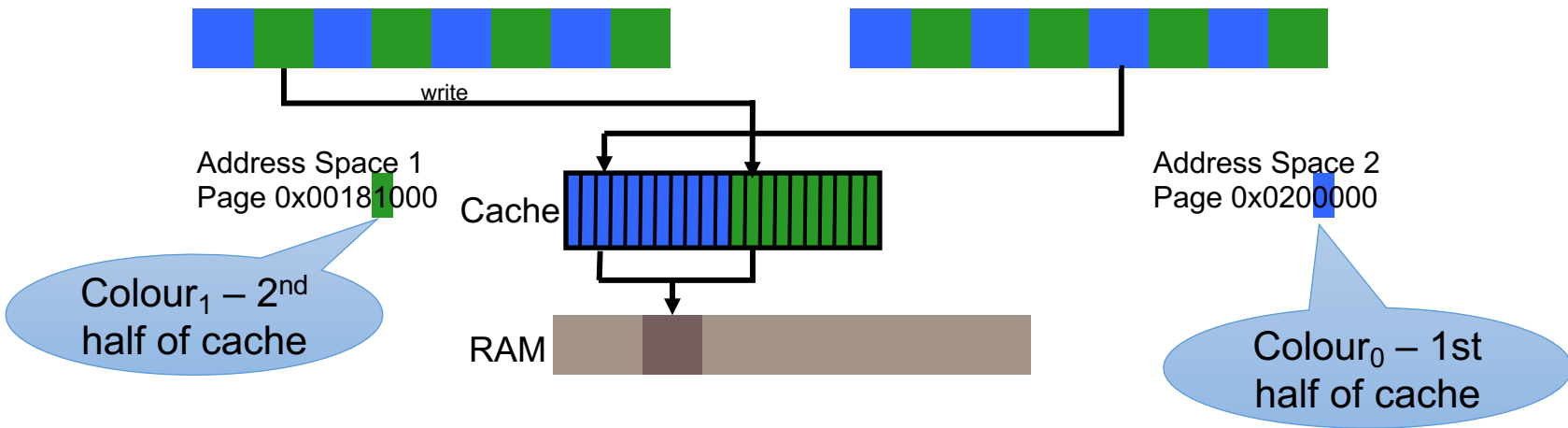
Virtually-Indexed Cache Issues: Aliasing

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 sets
 - ... 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 aliases a problem?
 - don't exist in single-colour cache
 - no problem for R/O data or I-caches

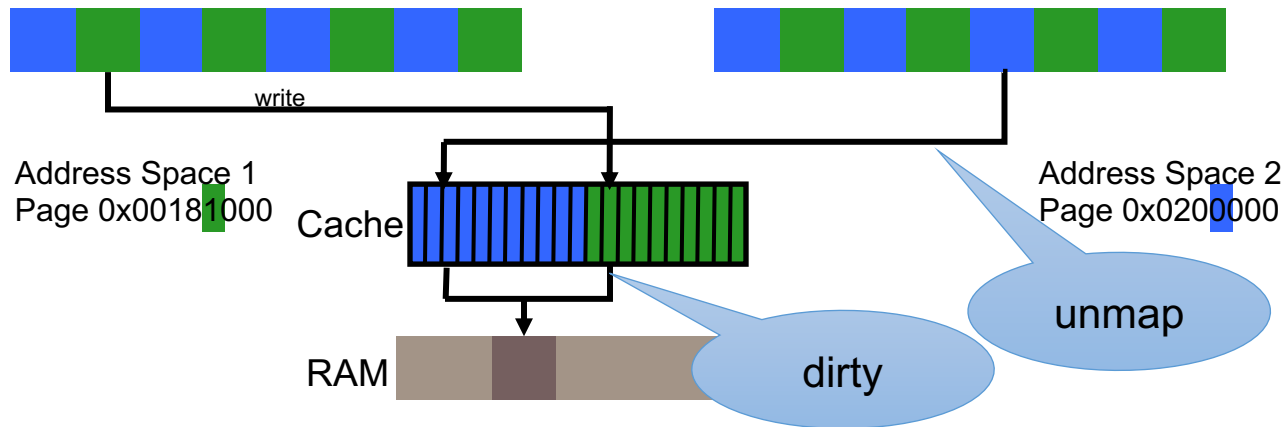


Aliasing Problem [1/2]



- 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*

Aliasing Problem [2/2]

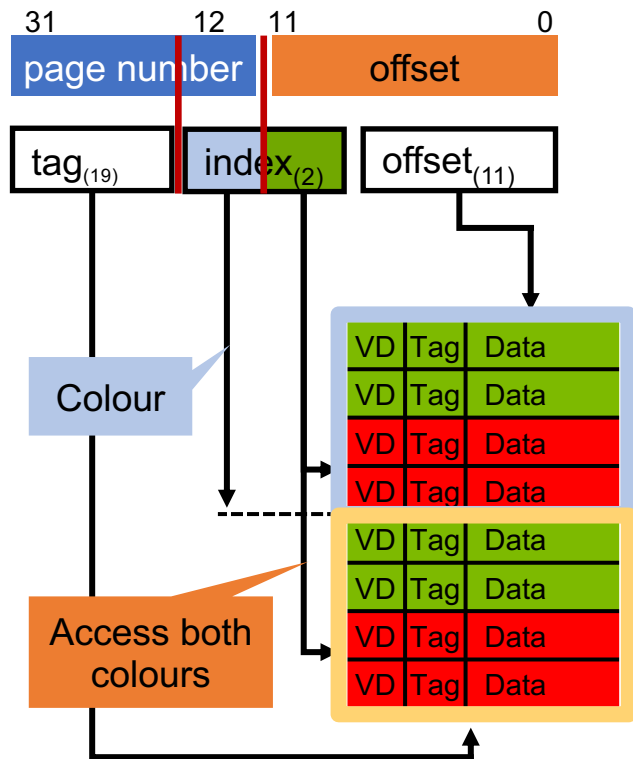


- Unmap aliased page, remaining page has a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Access new page
 - alias may write back after remapping: “*cache bomb*”

Avoiding Aliasing Problems

- Flush cache on context switch
 - doesn't help for aliasing *within* address space!
- Detect aliases and ensure:
 - all read-only, or
 - only one alias mapped
- Restrict VM mapping so all aliases are of the same colour
 - eg ensure $VA_{12} = PA_{12}$ – colour memory!
- Hardware alias detection

Hardware Alias Detection (Arm Cortex-A55)



Lookup accesses sets of both colours

- If the access misses in a set then lookup in aliases
 - If the access is a store then invalidate the alias of the “wrong” colour
- VP cache behaves like PP despite multiple colours!
- D-cache only, I-cache still needs invalidating all sets for self-modifying code!

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

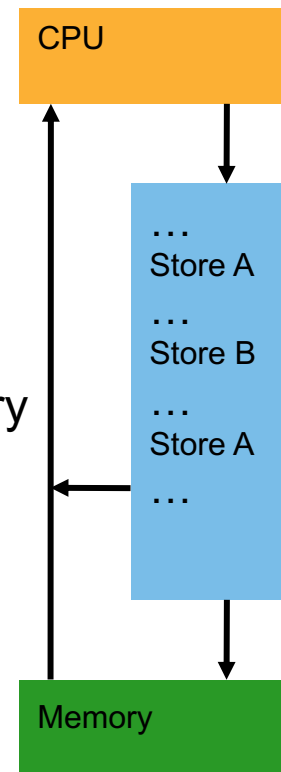
✓ Cache can use *bus snooping* for DMA/multicore coherency

✓ Obvious choice for L2–LLC where speed matters less

Cache Hierarchy

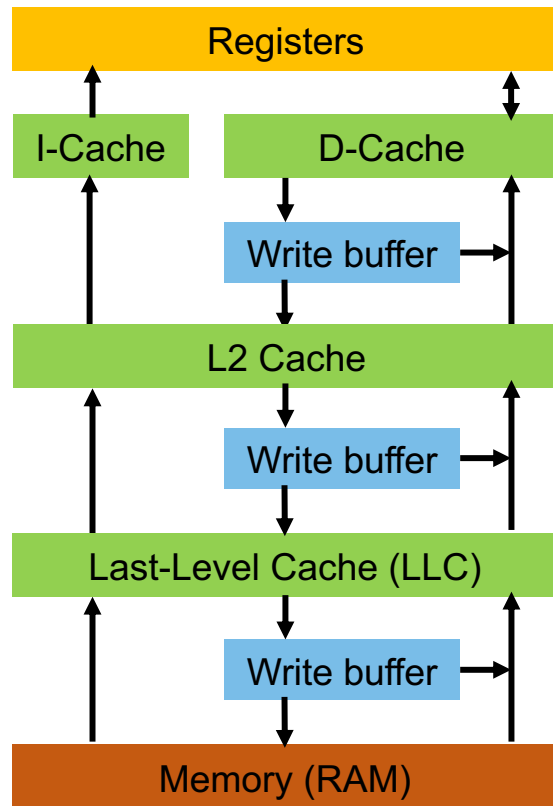
Write Buffer

- 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 multicore context

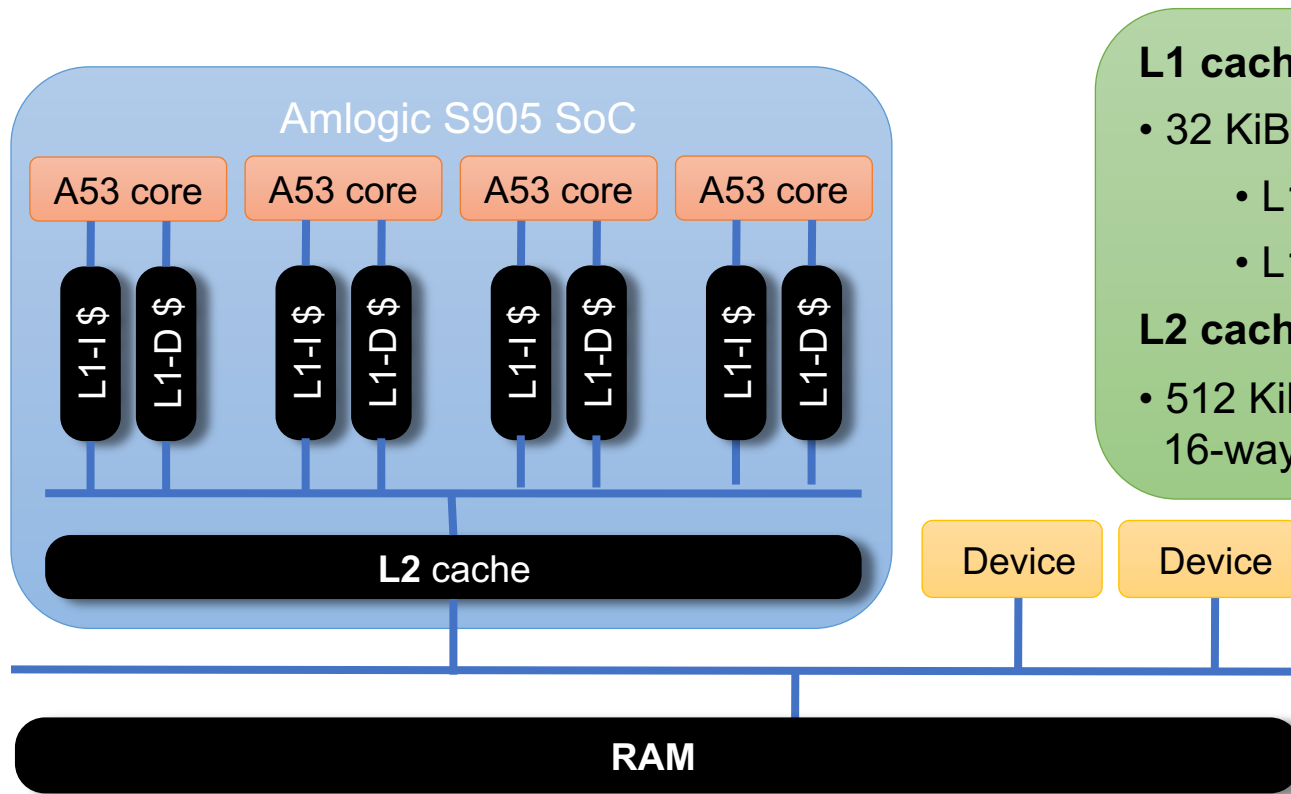


Cache Hierarchy

- 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-P, AMD Zen) or clustered (Intel-E, Arm Cortex)



ODROID-C2 (Cortex-A53) System Architecture



L1 cache:

- 32 KiB, 64-B lines
 - L1-I: 2-way, VP
 - L1-D: 4-way, PP

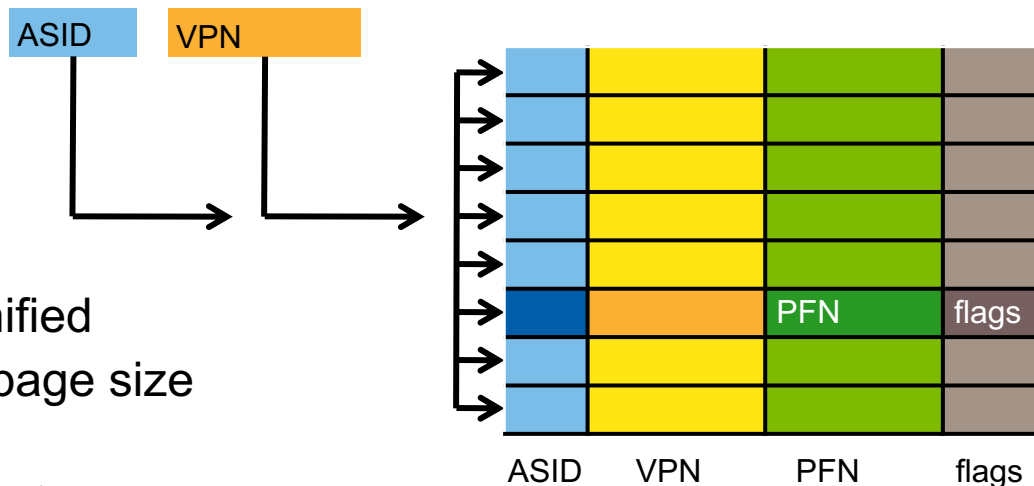
L2 cache:

- 512 KiB, 64-B lines
16-way, PP

TLB

Translation Lookaside Buffer (TLB)

- TLB is a (VV) cache for page-table entries
- TLB can be
 - software loaded, maintained by OS
 - hardware loaded, transparent to OS (standard these days)
- TLB can be:
 - split: I- and D-TLBs, or unified
 - partitioned: one for each page size
 - load/store-only
 - coalesced: a single TLB entry caching multiple translations



TLB Size

TLB coverage

- Memory sizes are increasing
- Number of TLB entries are roughly constant
- Base page sizes are steady
 - usually 4 KiB (SPARC V9, Alpha used 8 KiB)
 - 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!

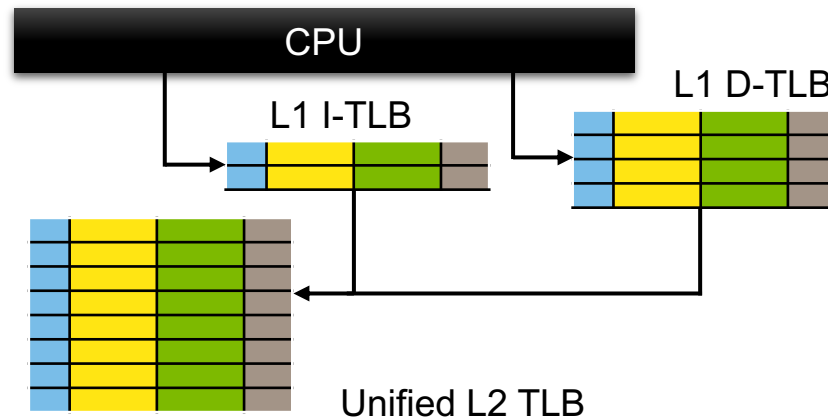
TLB can become a bottleneck!

Multi-Level TLBs

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

Intel Core Ultra Series 3 P-core (PTL-P)

L	I/D	Pages	Assoc	Entr
1	I	4 KiB	8-way	256
1	D-load	4 KiB	8-way	128
1	D-store	4 KiB – 1 GiB	fully	16
1	I	2/4 MiB	8-way	32
1	D-load	2/4 MiB	4-way	32
1	D-load	1 GiB	fully	8
2	unif	4 KiB, 2/4 MiB	8-way	2048
2	unif	4 KiB, 1 GiB	4-way	1024



Arm Cortex-A53

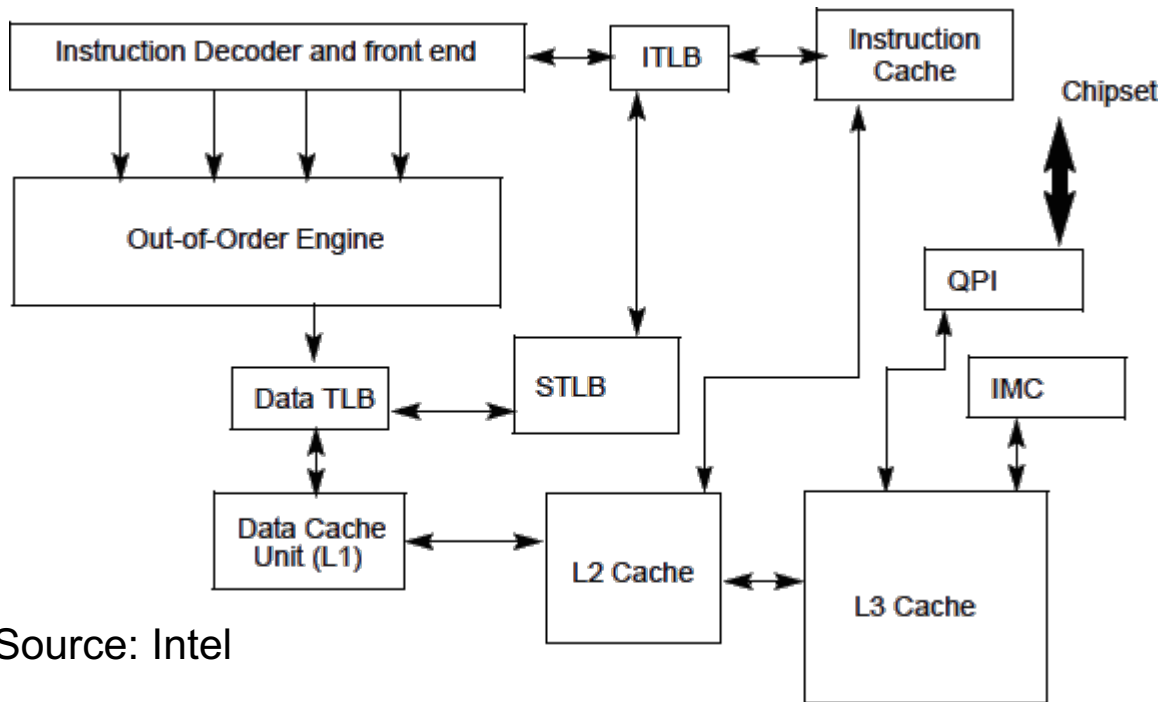
L	I/D	Pages	Assoc	Entr
1	I	4 KiB–1 GiB	fully	10
1	D	4 KiB–1 GiB	fully	10
2	unif	4 KiB–512 MiB	4-way	512

TLB Implementations

Not much growth in 50 years!

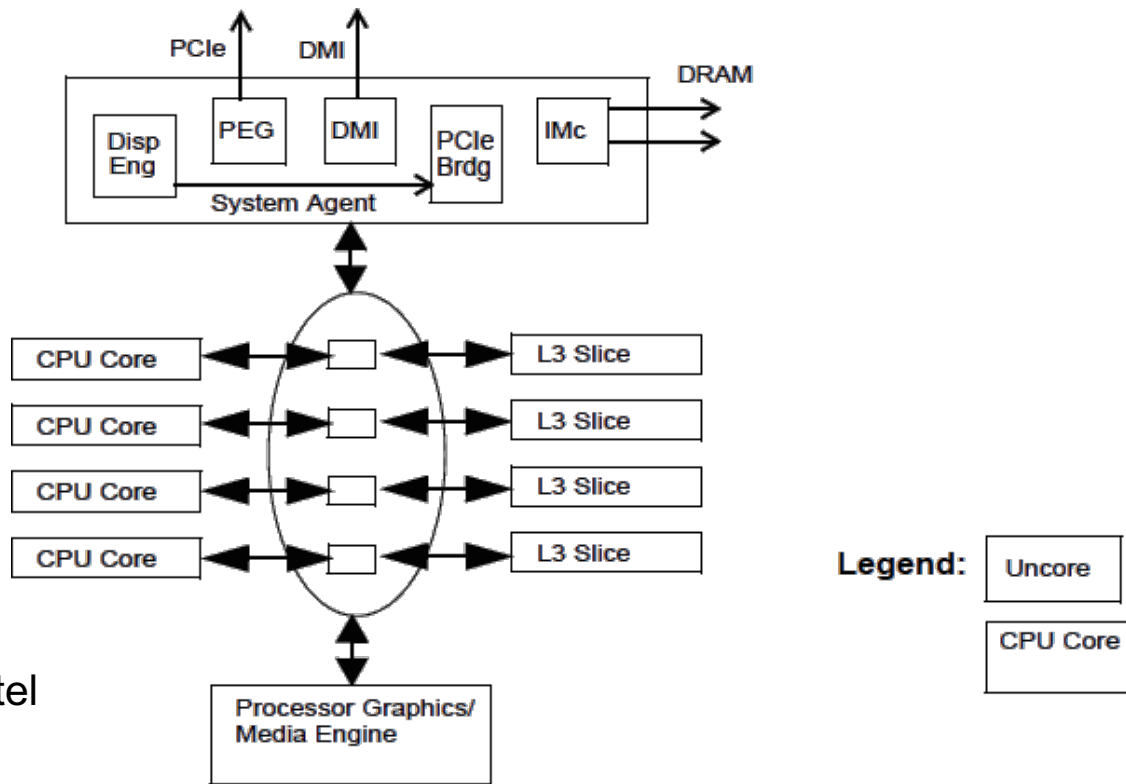
ISA (CPU)	Year	Size (I+D)	Assoc	Page Size	Coverage
VAX (VAX-11/780)	1977	64	2	0.5KiB	32 KiB
x86 (80386)	1985	32	4	4KiB	128 KiB
MIPS (R2000)	1986	64	full	4KiB	256 KiB
Alpha (21064)	1992	8i+32d	full	8KiB	256 KiB
PowerPC (601)	1993	4i+256d	2	4KiB–8MiB	1 MiB
SPARC (UltraSPARC)	1995	64i+64d	full	8KiB–4MiB	512 KiB
PA-RISC (PA-8000)	1996	96	full	4KiB–16MiB	384 KiB
IA-64 (Itanium)	2001	64i; L1:32d; L2:96d	full	4KiB–4GiB	384 KiB
ARMv7 (Cortex-A9)	2007	L1:64i+32d; L2:512	2	4KiB–16MiB	2 MiB
Armv8 (Cortex-A53)	2012	L1:10i+10d; L2:512	4	4KiB–512MiB	2 MiB
x86-64 (PTL-P)	2026	L1:256i+128d*; L2:3072	12	4KiB–1GiB	12 MiB
Armv9 (C2-Ultra)	2026	L1:128i+96d; L2:2048	8	4KiB–1GiB	8 MiB
RISC-V (Ascalon S)	2026	L1:?i+256d; L2:1024	?	4KiB-1GiB?	4 MiB

Intel Core i7 (Haswell) Cache Structure



Source: Intel

Intel Haswell L3 Cache



Source: Intel

Peripheral Devices

Background: The Memory Contract [1/2]

Programmer's model of memory:

```
loadi r1, <addr>
loadi r0, <val>
store r0, r1      // store <val> at <addr>
...
load r2, r1       // r2 now contains <val>
```

Memory contract:

A read will return the last value written

Note: with shared memory, the last value written may be from someone else!

Background: The Memory Contract [2/2]

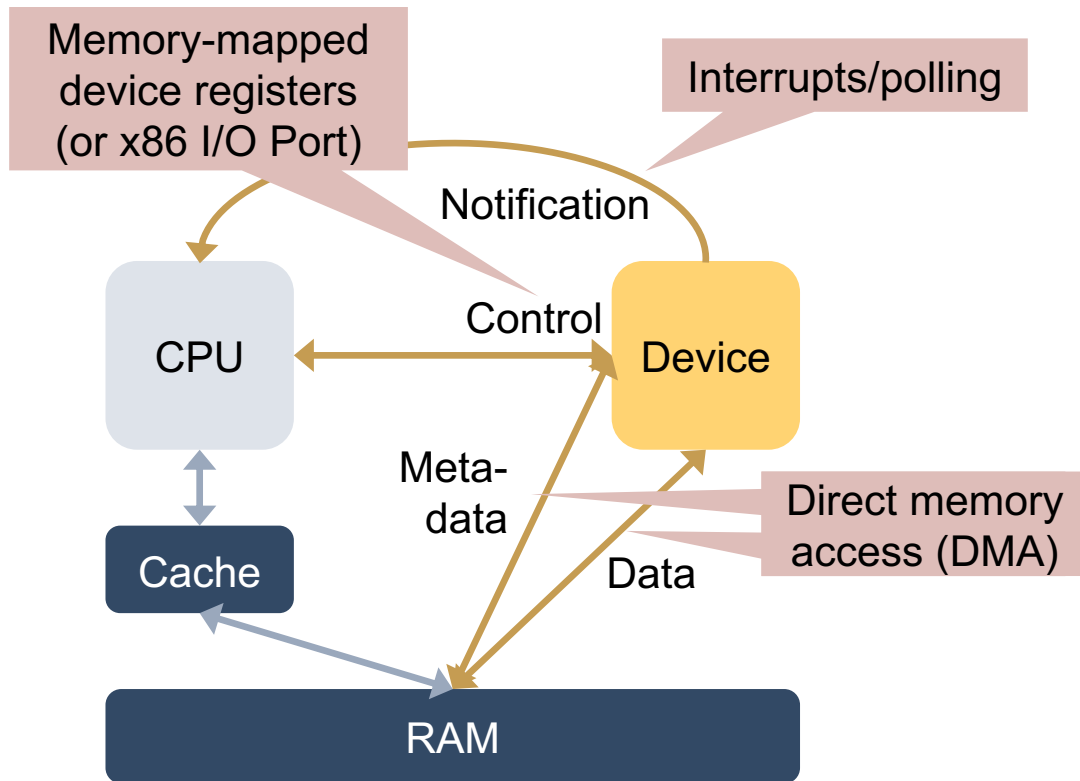
Programmer's model of memory:

```
int32_t *ip = <addr>;  
int32_t i = *ip;  
char *cp = (char*) ip;  
int32_t j = (cp[0] << 24) | (cp[1] << 16) | (cp[2] << 8) | cp[3];  
  
// now i==j, assuming big-endian
```

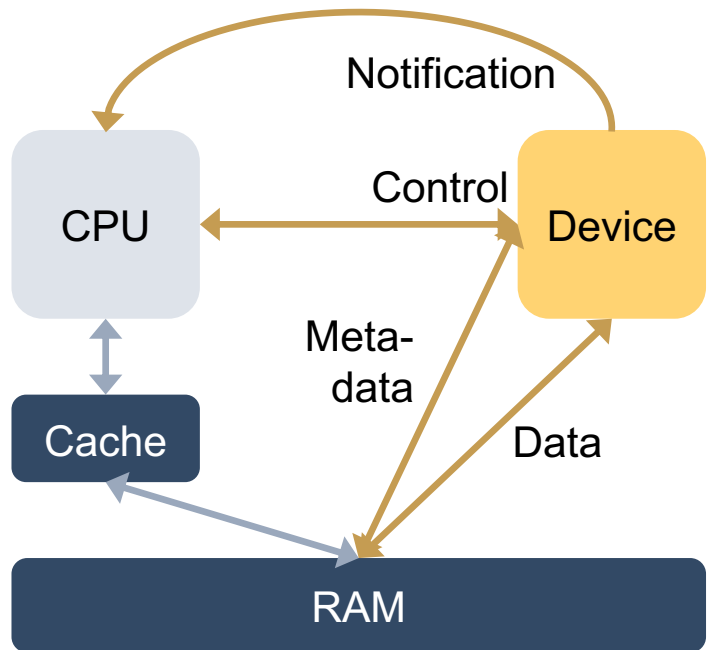
Memory contract:

Order or granularity of access don't matter

Peripheral Devices



Device-Access Caveats

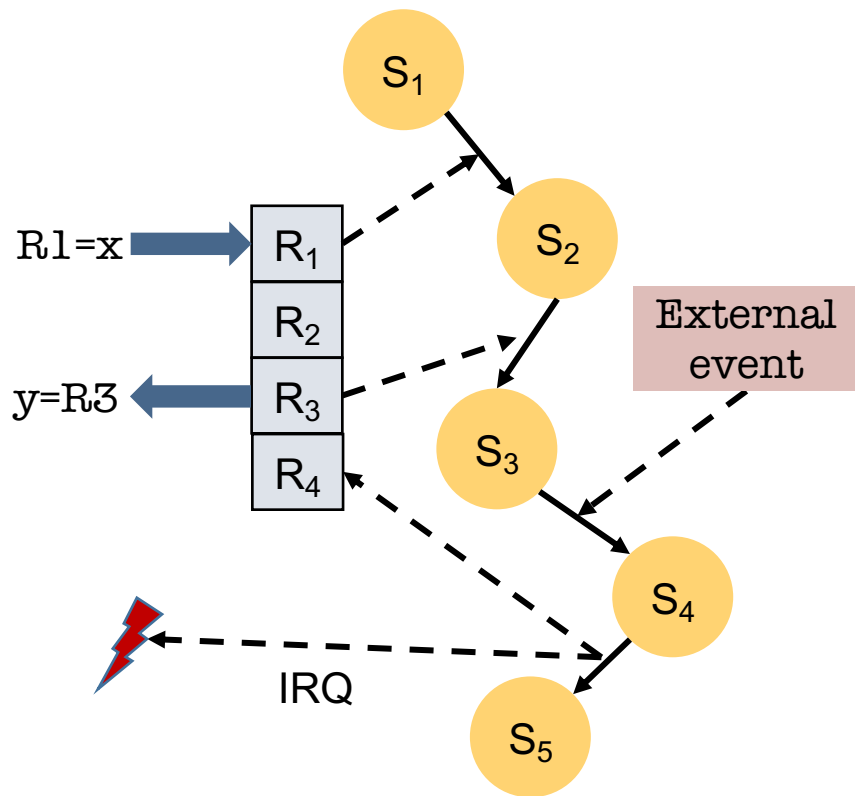


Device access bypasses cache!

- Device registers must be mapped uncached
- DMA buffers must be flushed/invalidated before initiating I/O
- Else:
 - write stale data
 - read data overwritten by old data (cache bomb!)

x86 keeps DMA cache-coherent; opt. on Arm and RISC-V

Devices Are State Machines



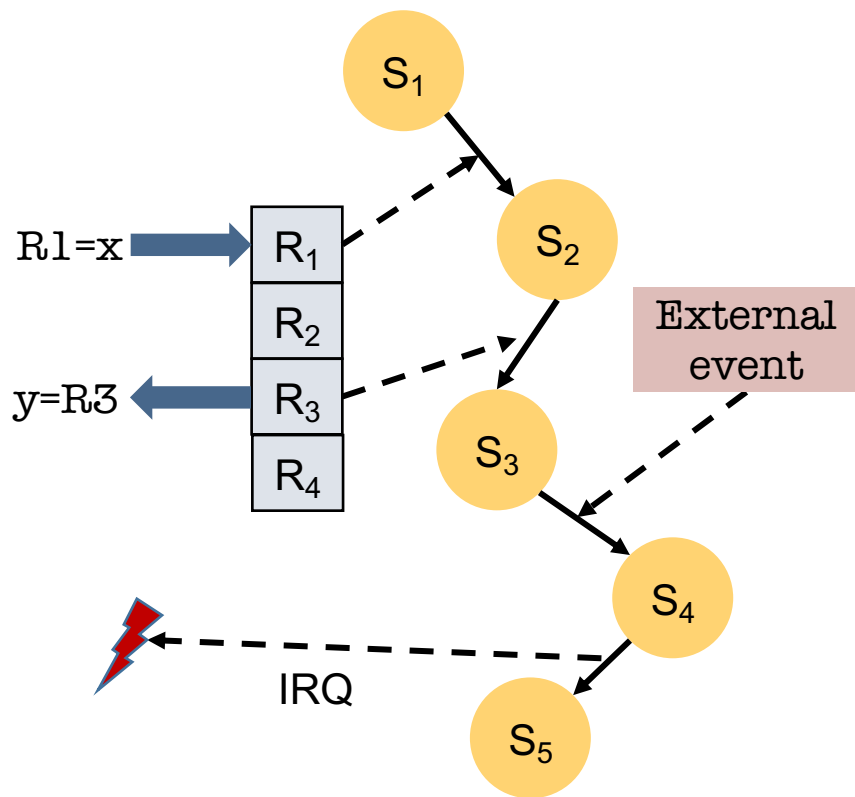
State transitions triggered by:

- Device register access
 - write to device register
 - read from device register
- External events
 - data available
 - transmit complete ...

State transitions:

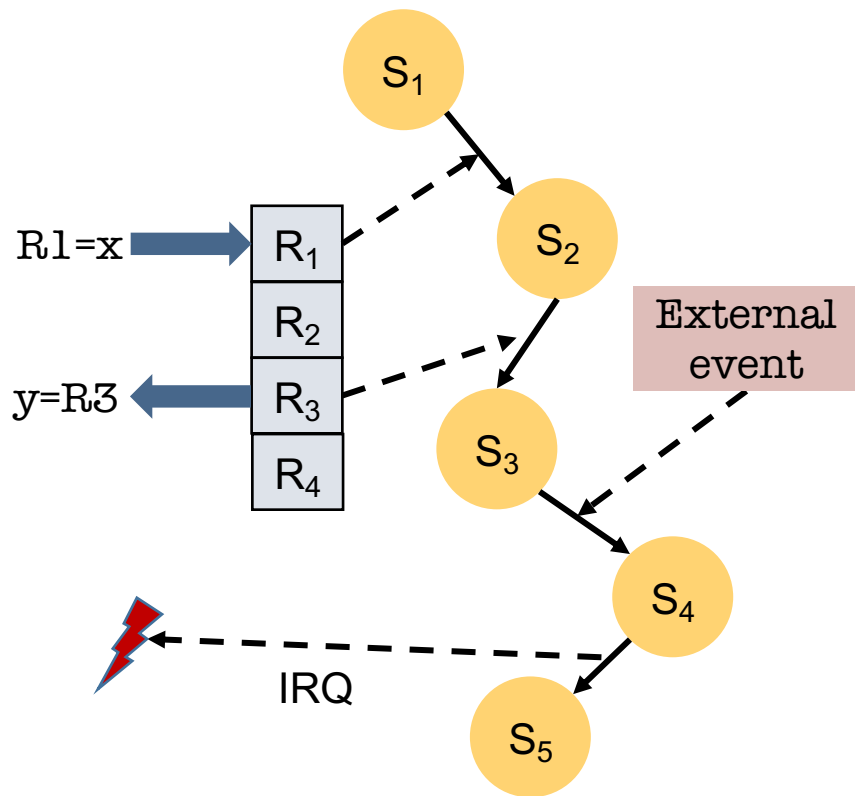
- Change register content
- Raise IRQs

Implication: Device Registers Aren't Memory!



- Writing same value twice may have different effects
- Reading same register twice may return different values
- Reading after writing:
 - may return different value
 - may trigger error
- Result of access may depend on elapsed time
- Reading 4 bytes is different from reading one int32
 - ... and may result in error

Device Protocol Examples



1. write char to R_1
2. wait 10 ms
3. read int32 from R_3
4. wait for IRQ or poll R_4 for $\neq 0$
5. ...

Device-specified or bus latency

Specified in device data sheet
... which is usually full of errors