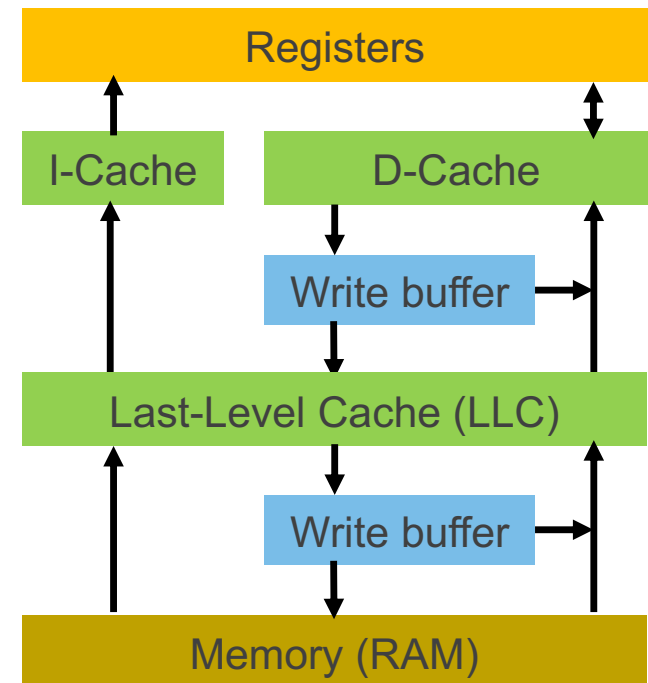


2024 T3 Week 02 Part 1  
**Hardware Considerations:**  
**What Every OS Designer Must Know**  
@GernotHeiser



# 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

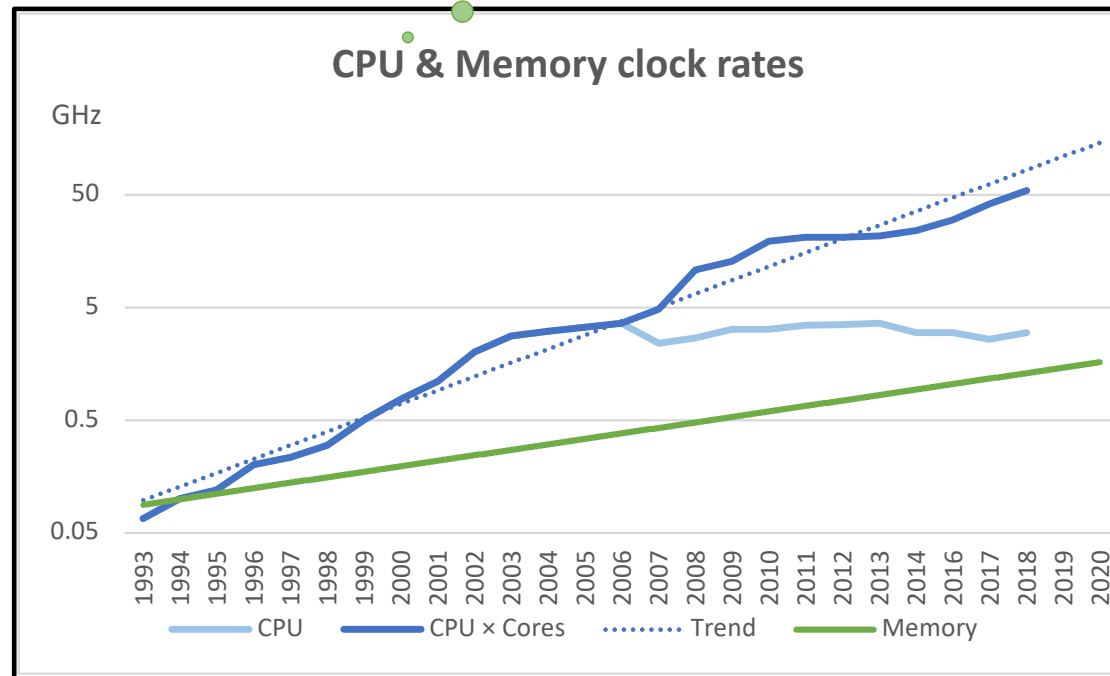
- Caches
  - What are caches, why do we have them?
  - How do they work (in detail)?
  - Why you need to understand them? – Software effects
  - Cache hierarchy
  - Translation caches: TLB
- Devices

**Later:** Concurrency effects and memory models

# Cache Basics

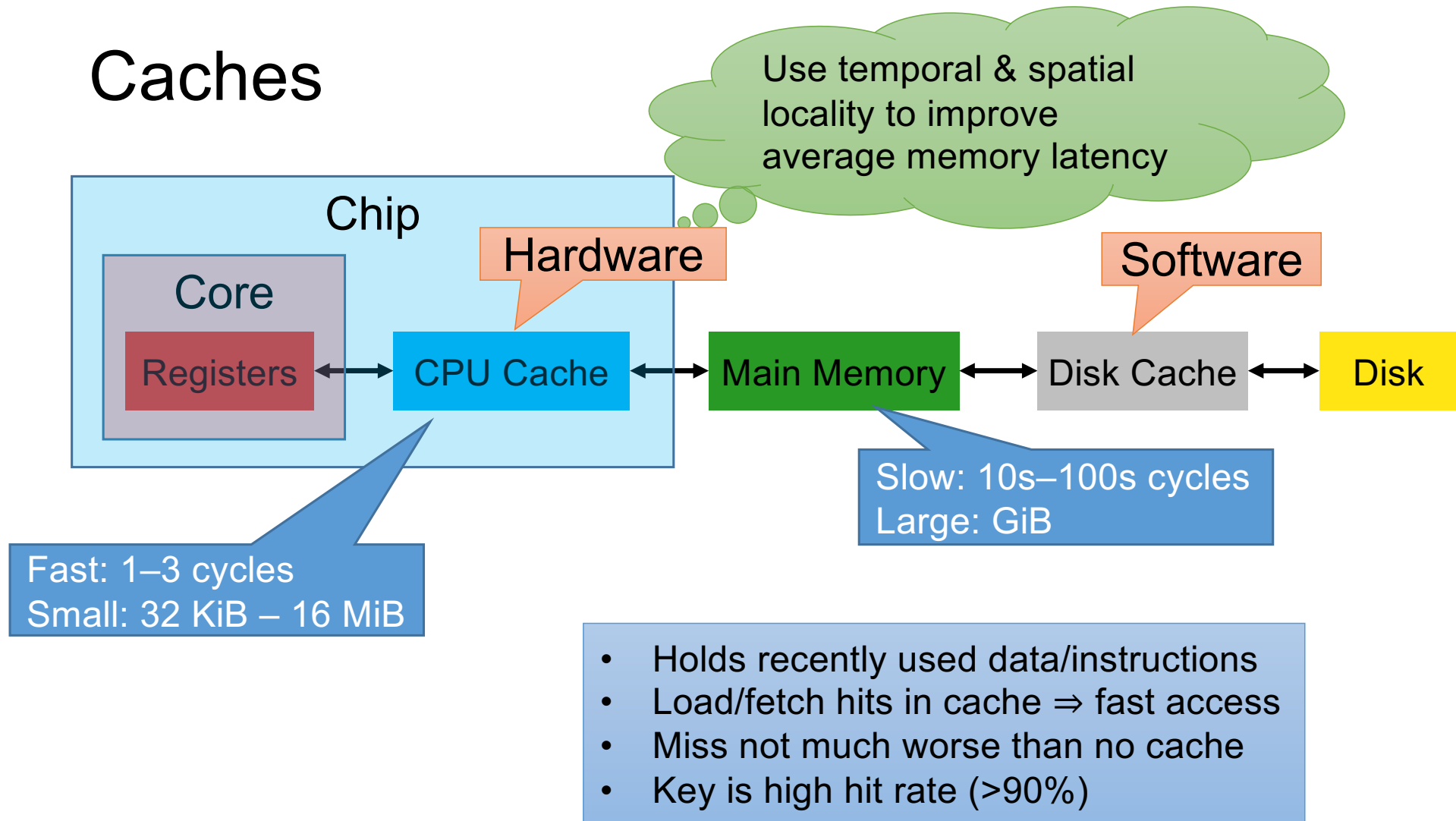
# The Memory Wall

Clock rates of Intel processors



Speed gap still widens by approx 18% per year!

# Caches



# Cache Organisation: Unit of Data Transfer



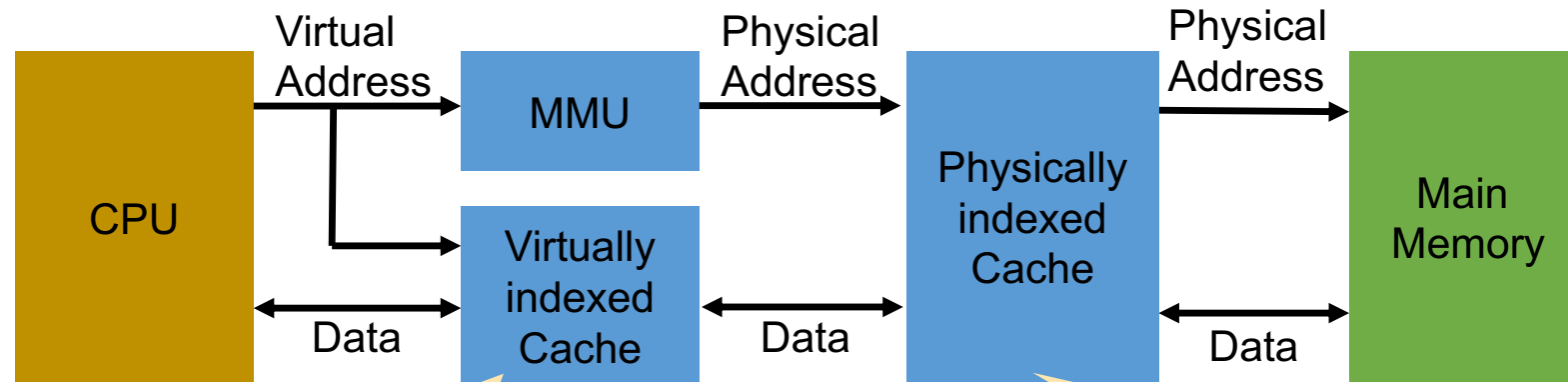
Line is also unit of allocation, holds data and

- 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

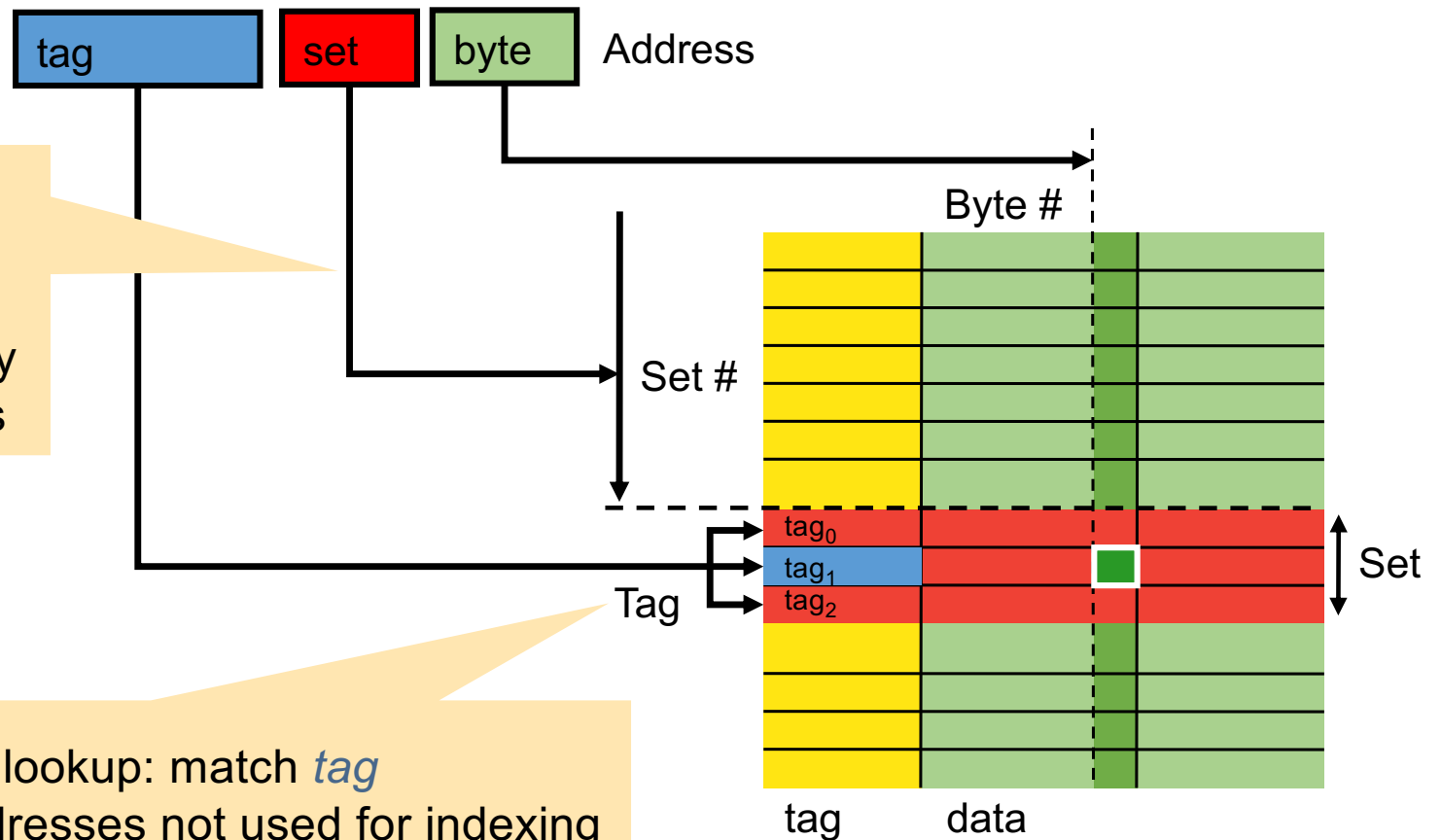
- L1 closest to CPU
- LLC: last-level cache
- Only L1 may be virtually addressed



# Indexing

# Cache Indexing

- 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

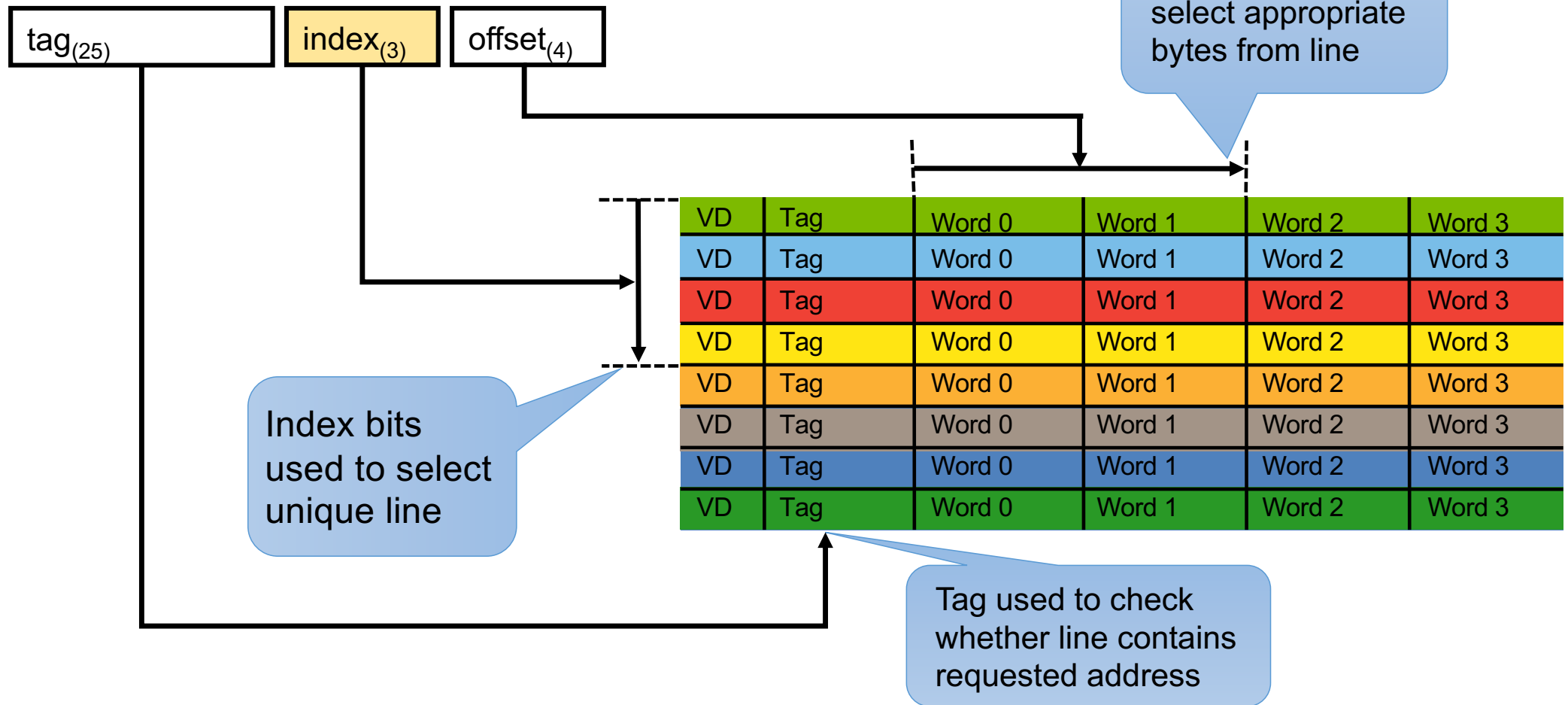


Many conflicts  
⇒ low hit rate

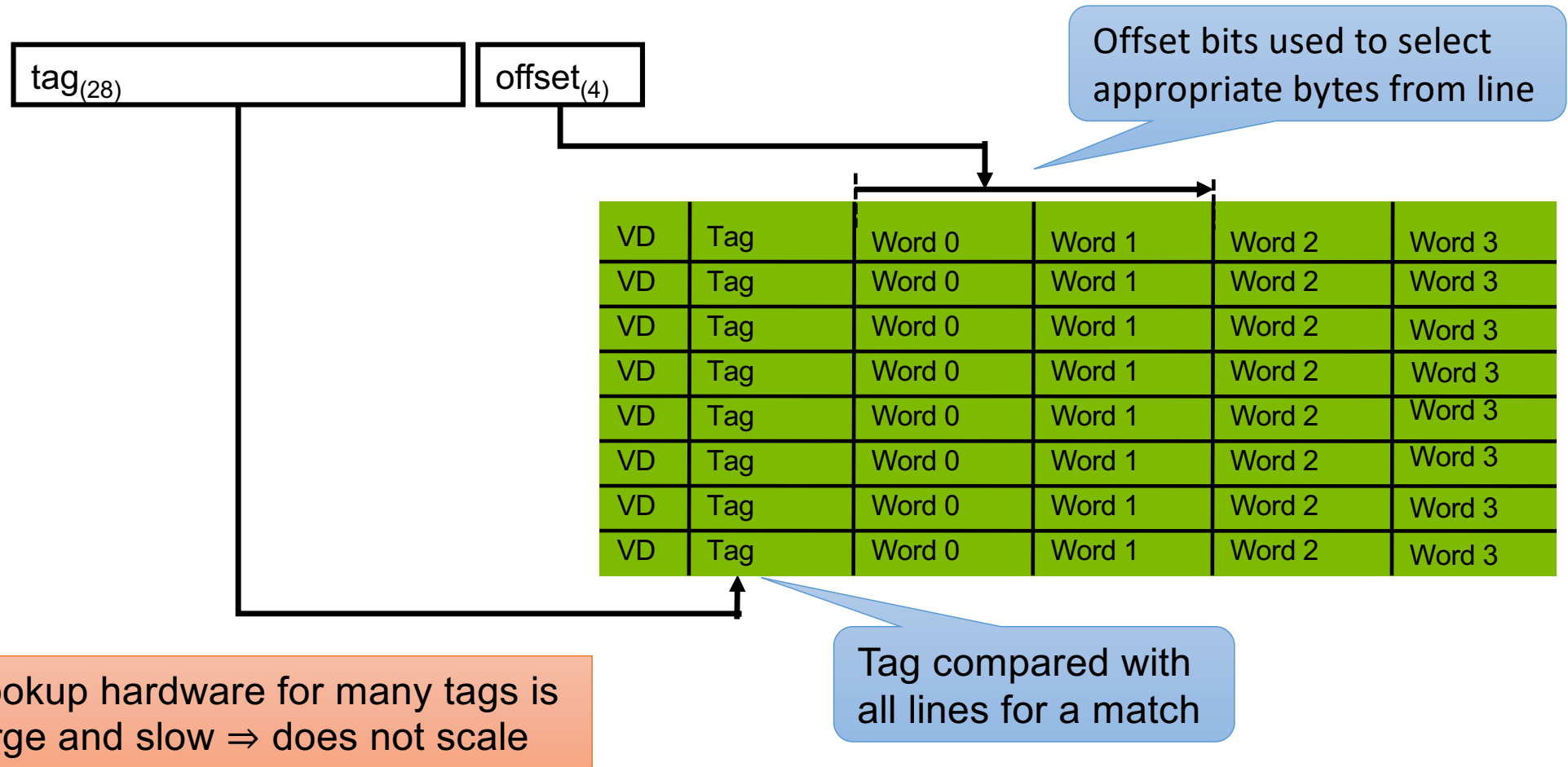
- $n$  lines per set:  $n$ -way set-associative cache
- $n = 1$ : *direct mapped*
- $2 \leq n < \# \text{ lines}$ : *set associative*
- $n = \# \text{ lines}$ : *fully associative*

Slow & power-hungry

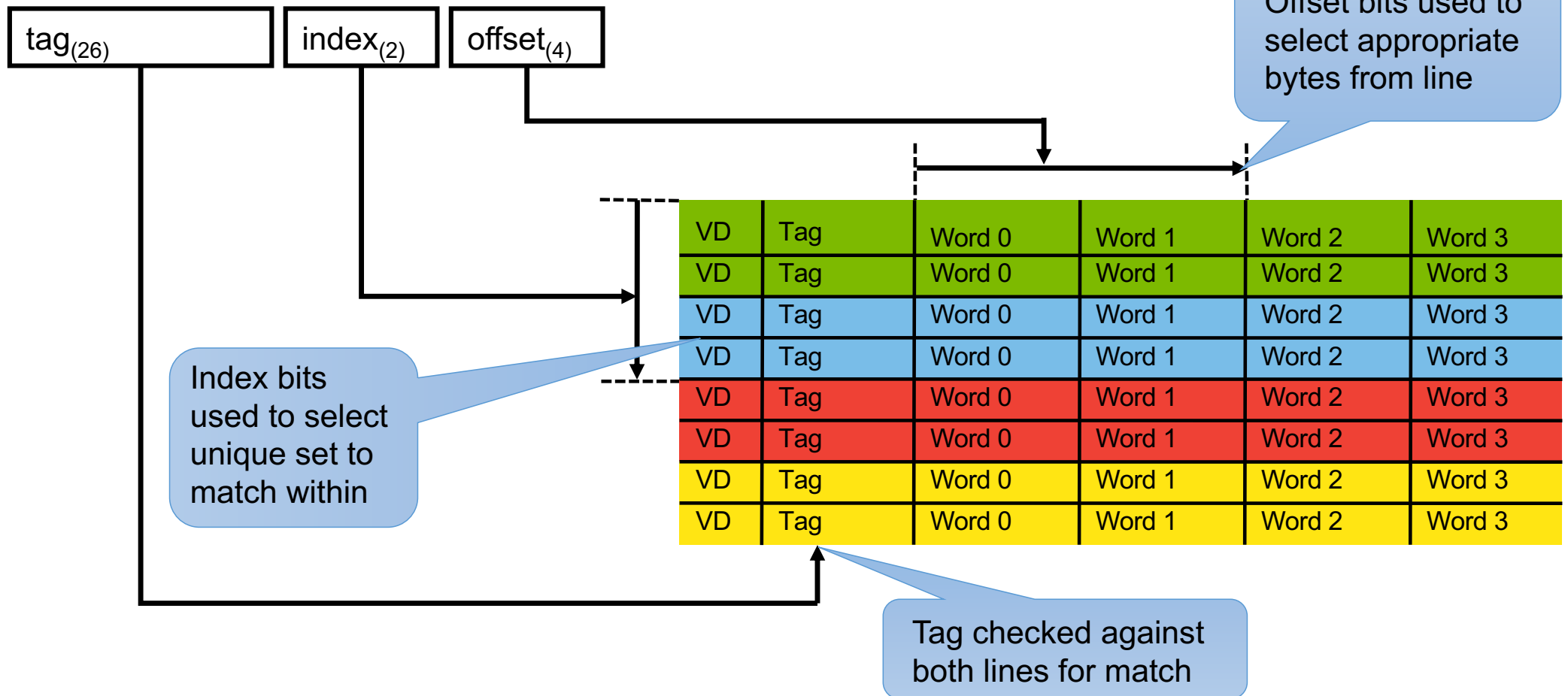
# Cache Indexing: Direct Mapped



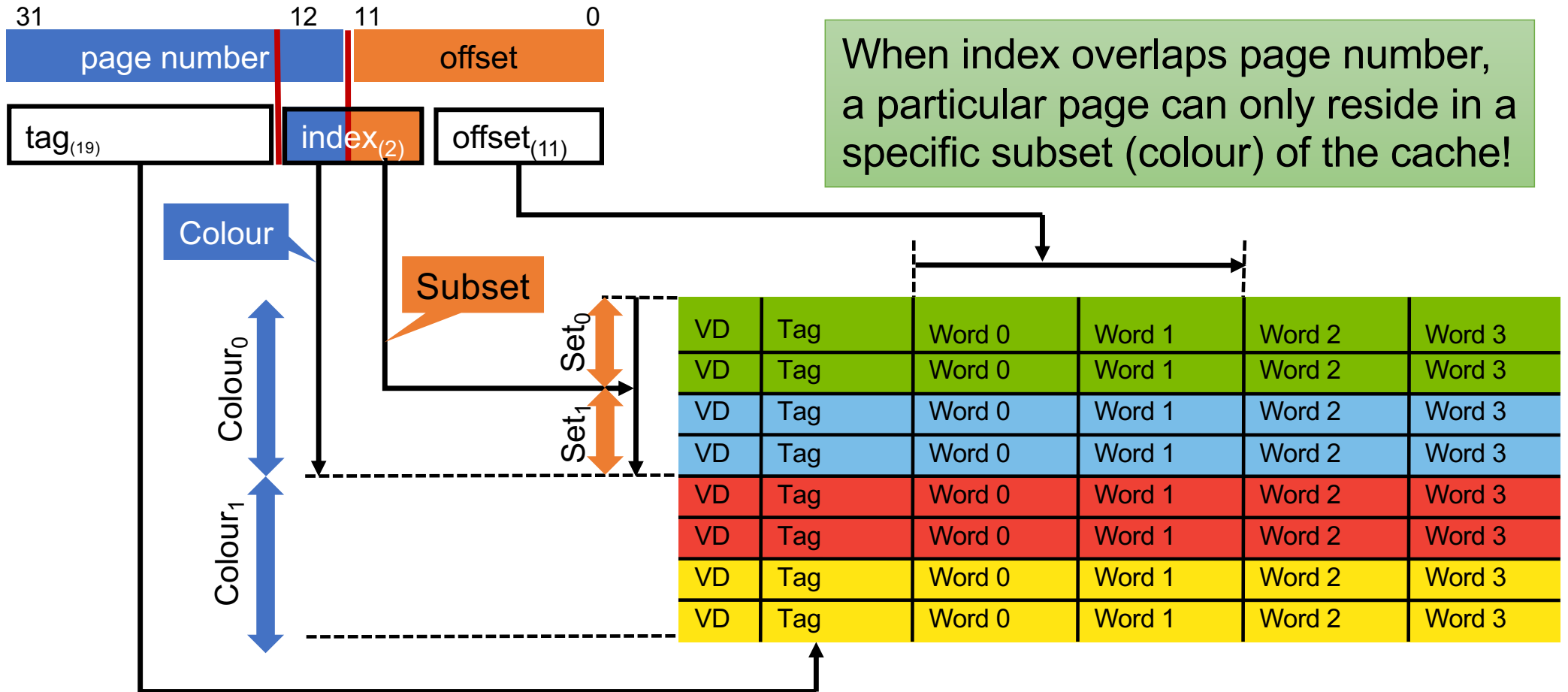
# Cache Indexing: Fully Associative



# Cache Indexing: 2-Way Associative

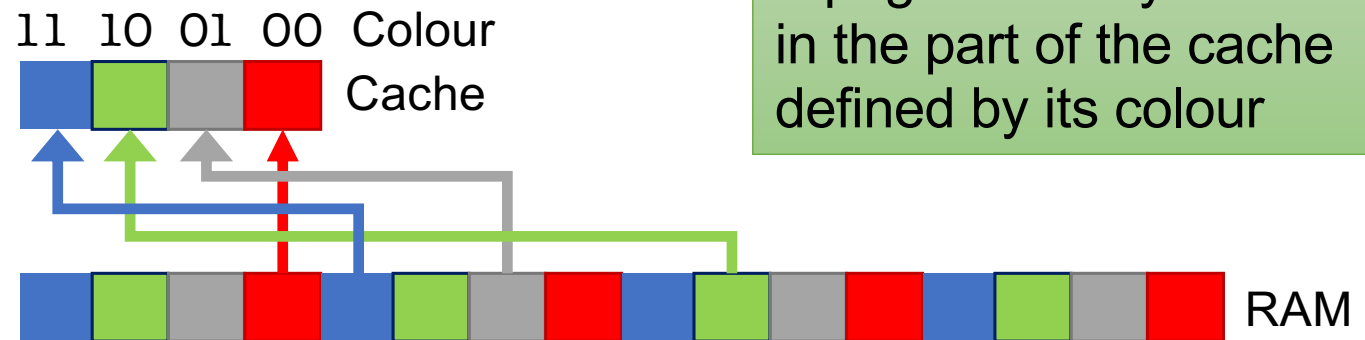


# Cache Associativity vs Paging



# 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

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

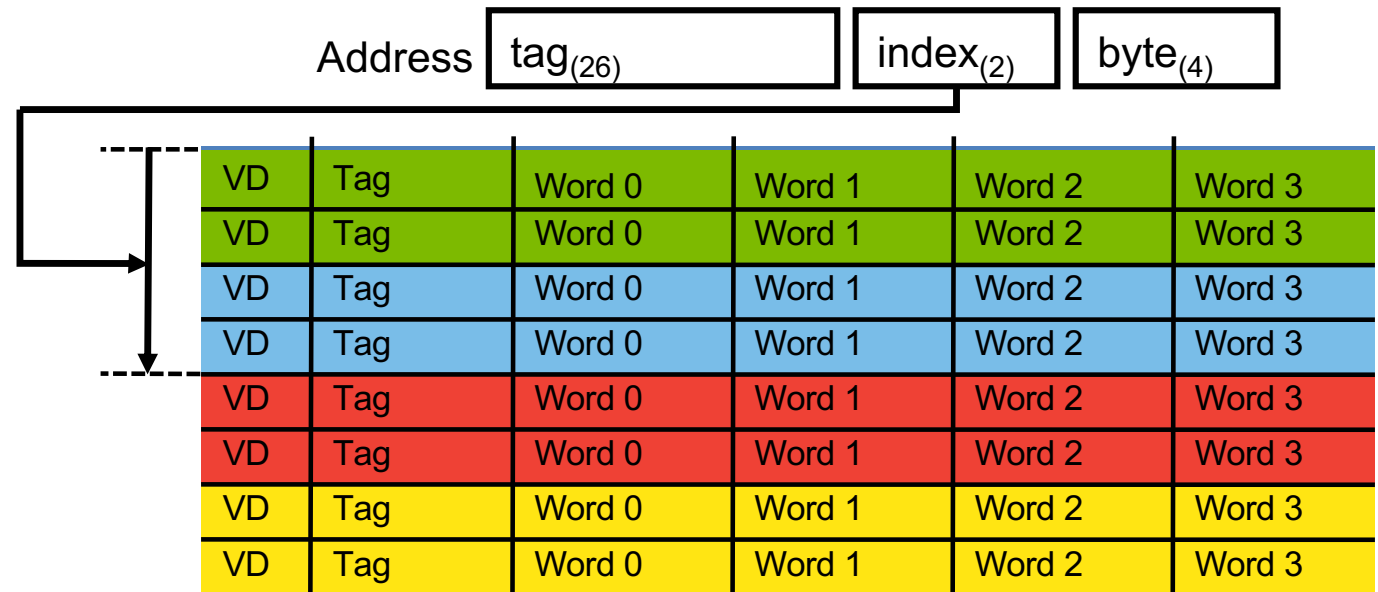
- *n-way* associative cache can hold *n lines* with the same *index* value
- More than *n lines* are competing for same index forces a miss!
- There are four different types of cache misses (“*the four Cs*”):
  - **Compulsory miss**: data cannot be in the cache (of infinite size)
    - First access (after 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 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
  - **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-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

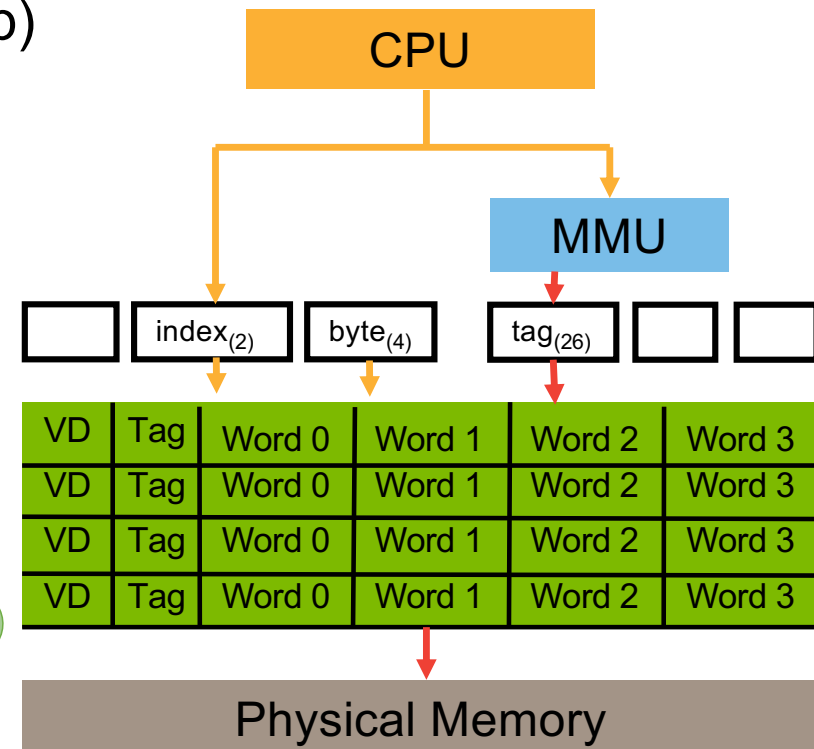
Rare these days,  
will ignore

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



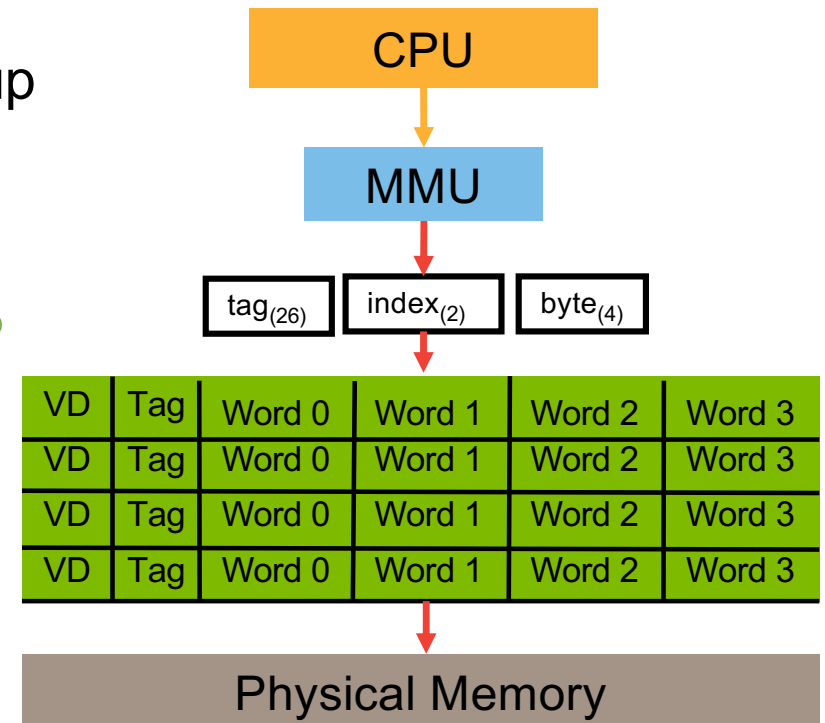
# Physically-Indexed, Physically-Tagged Cache

- 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!**

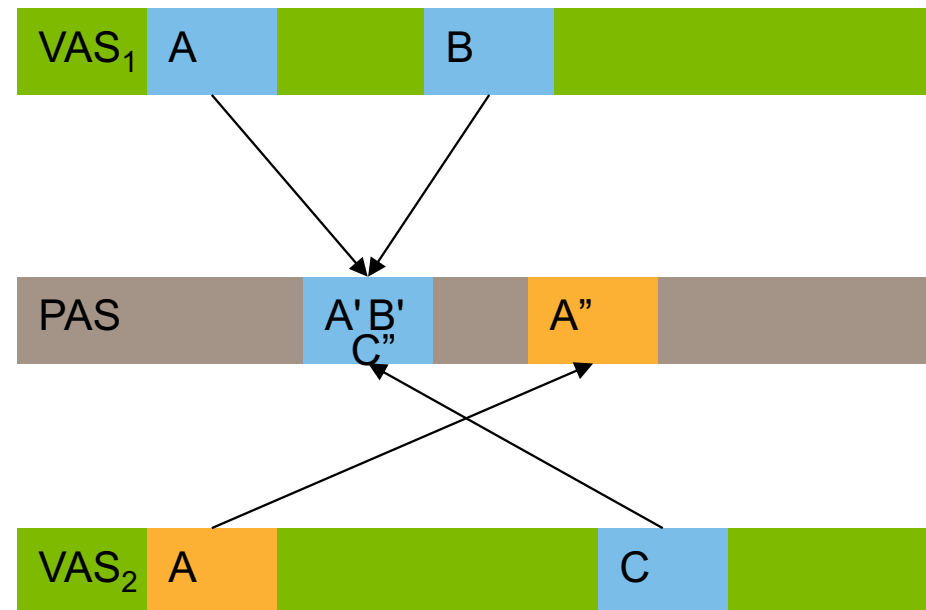




# Software-Visible Effects

# Cache Issues

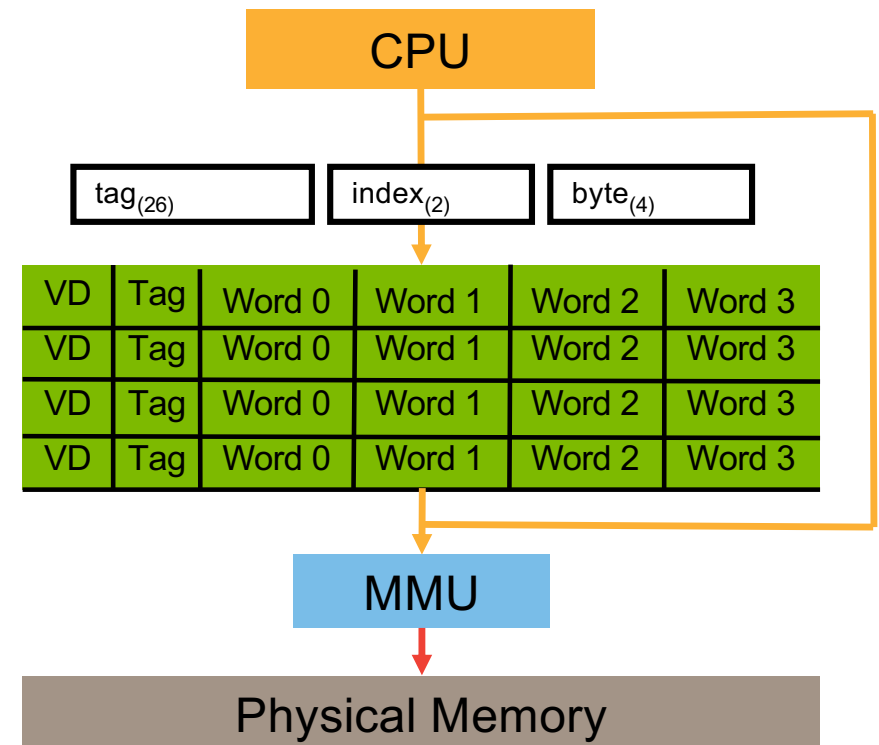
- Caches are managed by hardware transparently to software, so OS doesn't have to worry about them, ~~right?~~ **Wrong!**
- Software-visible cache effects:
  - *performance*
    - cache-friendly data layout
  - *homonyms*:
    - same address, different data
    - can affect correctness!  
(on VV caches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



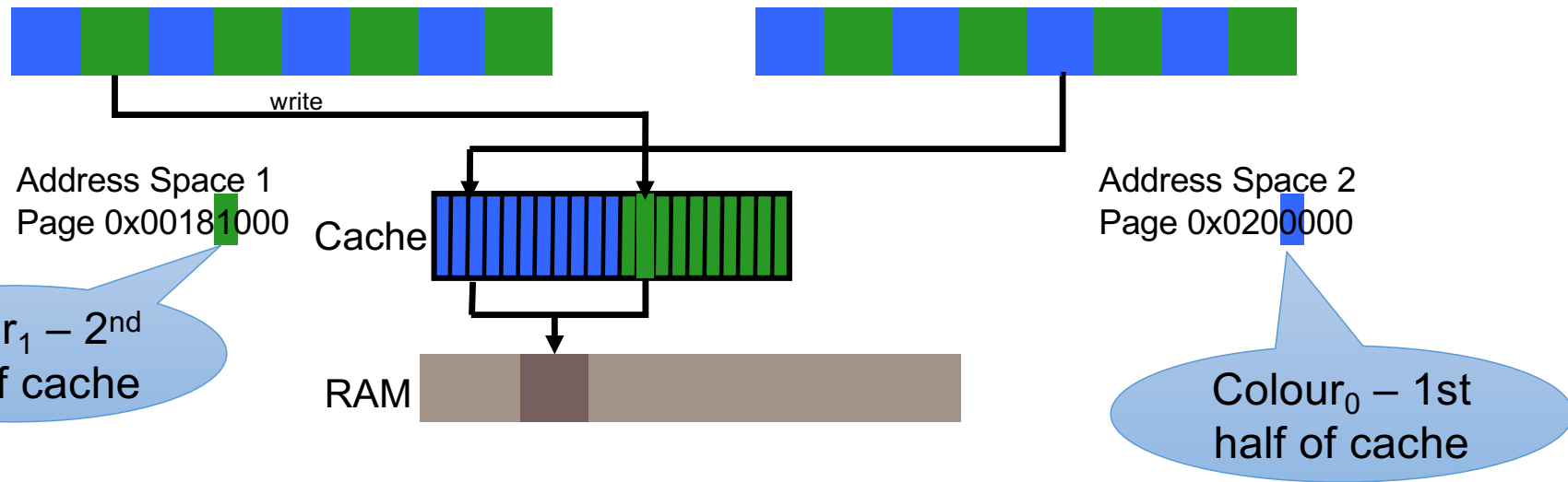
# Virtually-Indexed Cache Issues: Aliasing

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

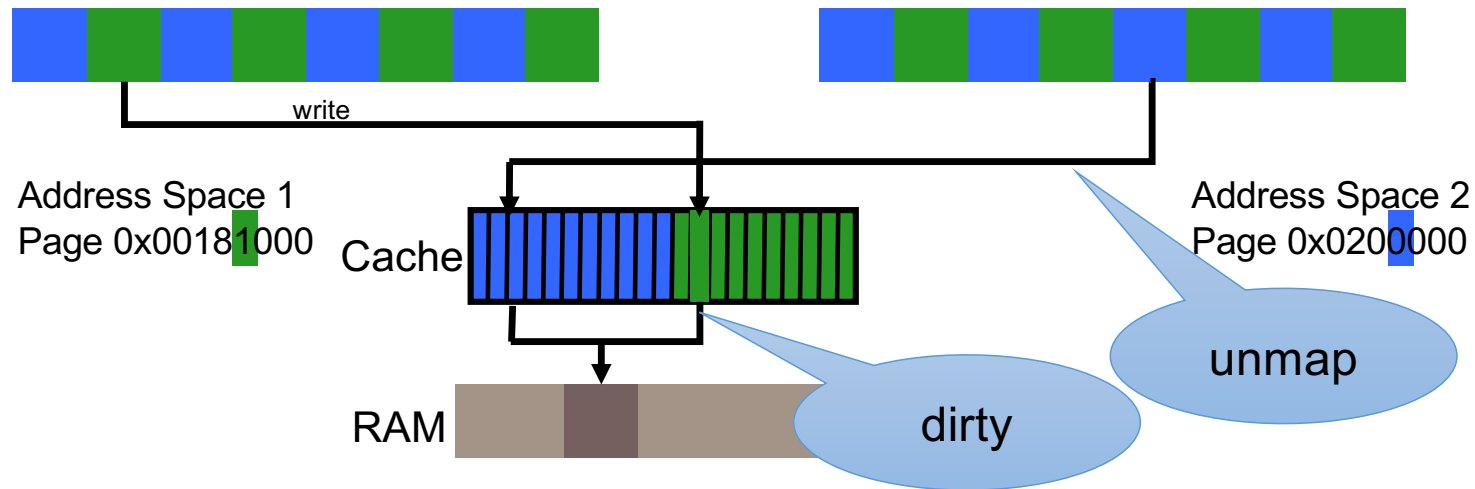


# Aliasing Problem [1/2]



- 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]

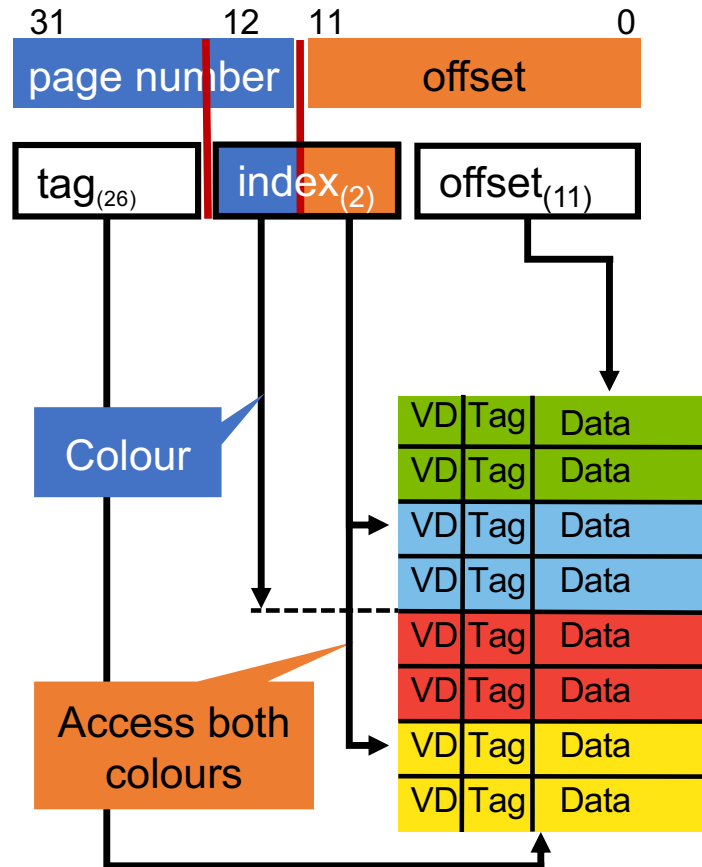


- Unmap aliased page, remaining page has a dirty cache line
- Re-use (remap) frame for a different page (in same or different AS)
- Access new page
  - alias may write back after remapping: “*cache bomb*”

# 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 A53)



Lookup accesses sets of both colours

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

# 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

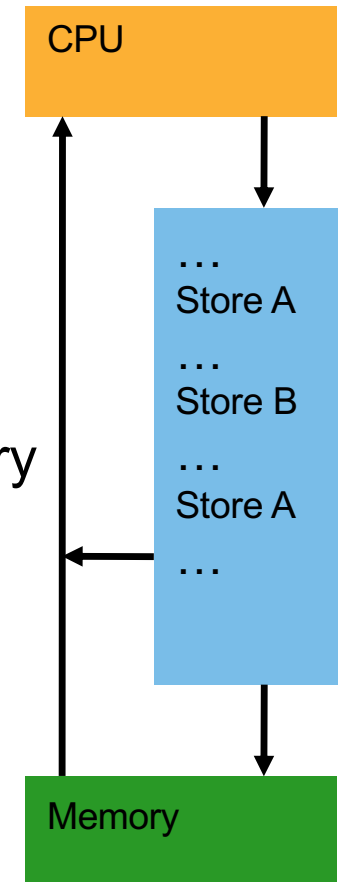
✓ Cache can use *bus snooping* for DMA/multicore coherency

✓ Obvious choice for L2–LLC where speed matters less

# Cache Hierarchy

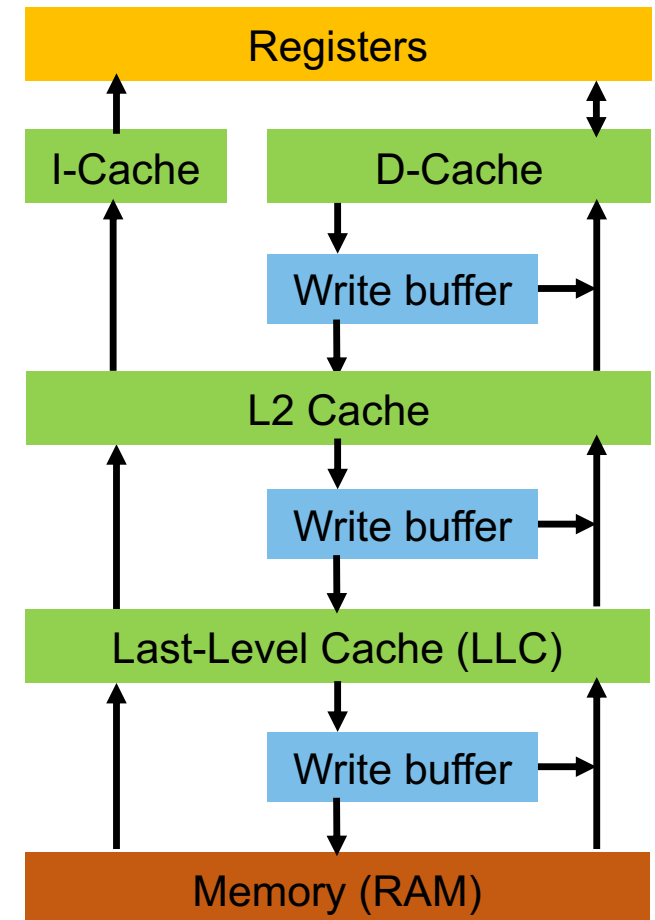
# Write Buffer

- 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 SMP context

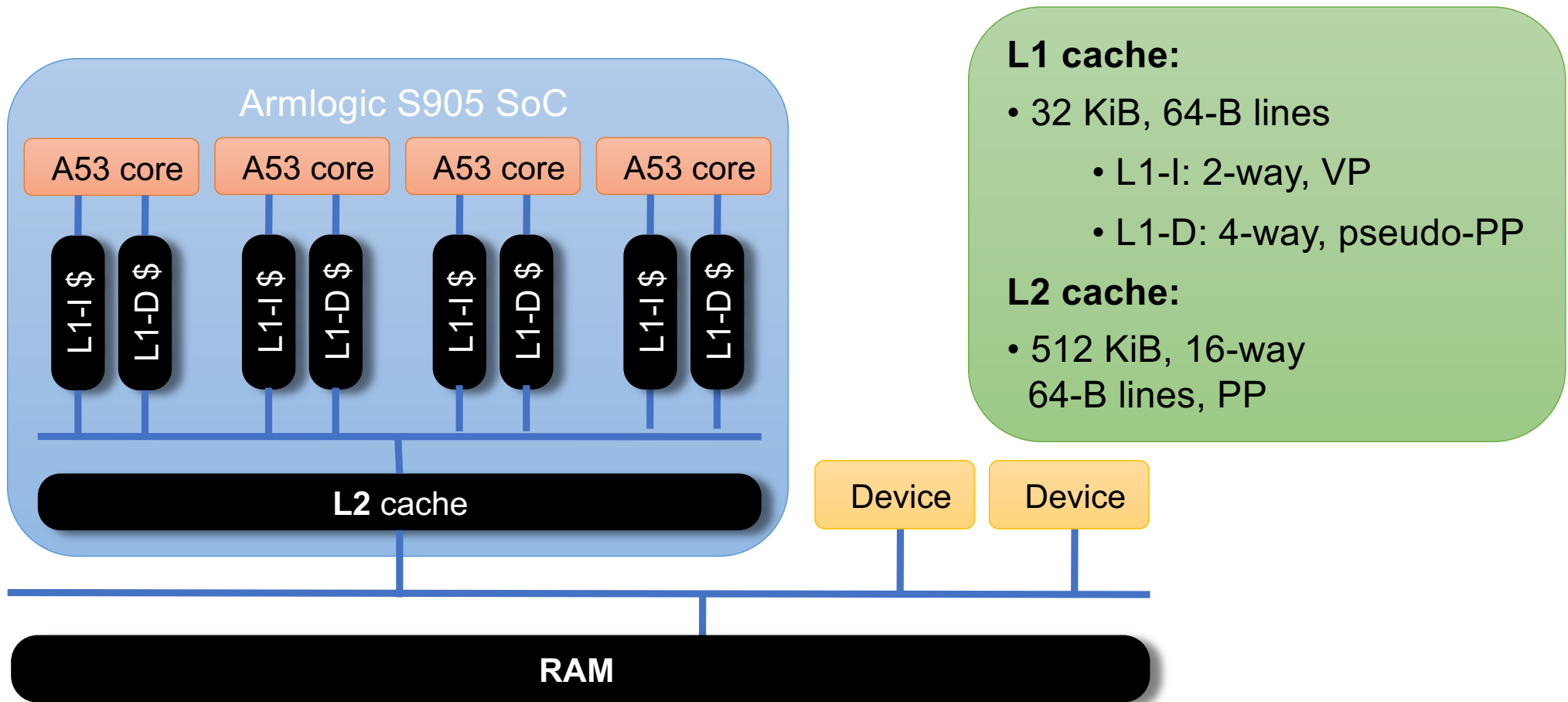


# Cache Hierarchy

- 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)



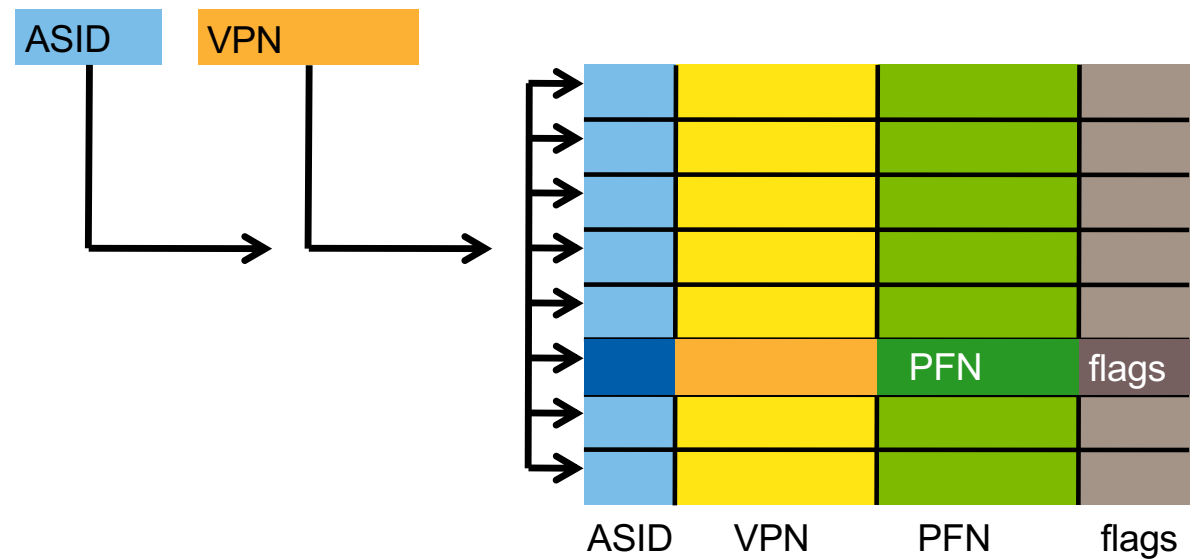
# ODROID-C2 (Cortex A53) System Architecture



# 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
  - 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

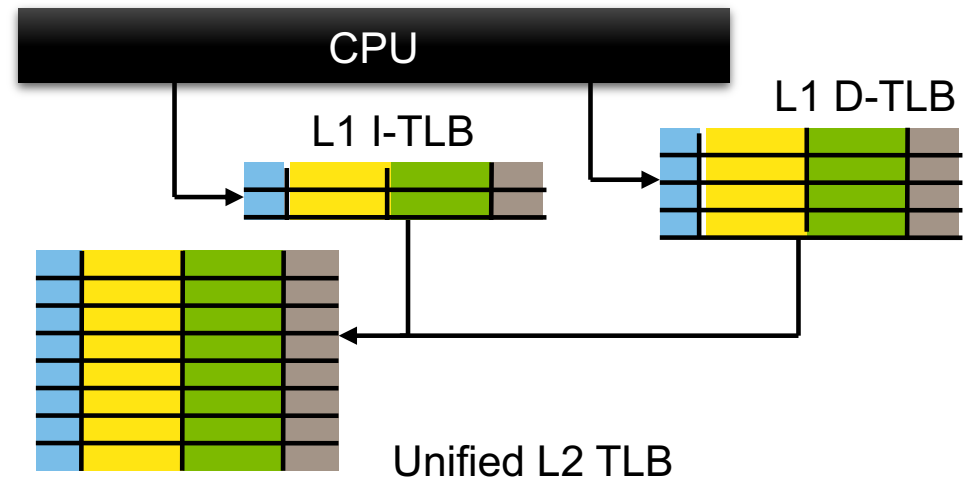
## TLB coverage

- 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



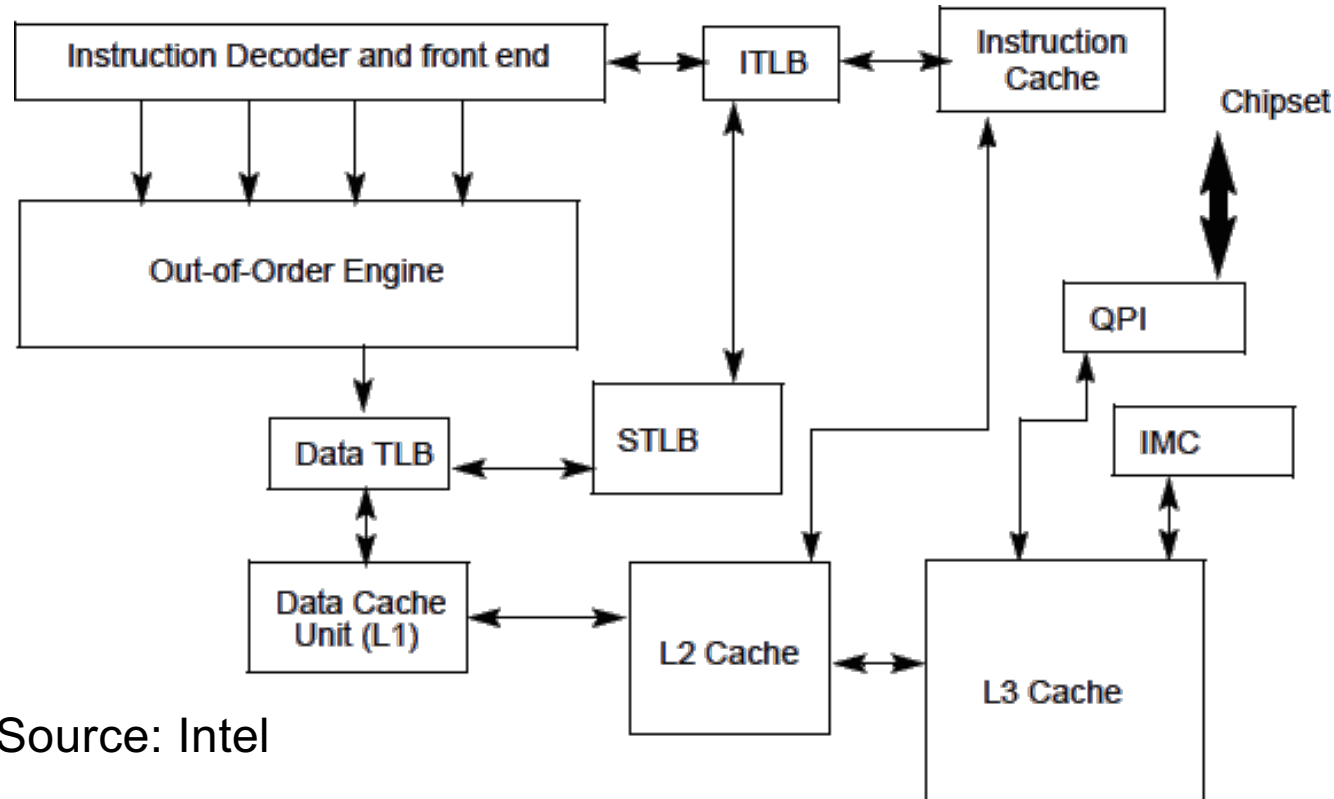
Intel Core i7

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

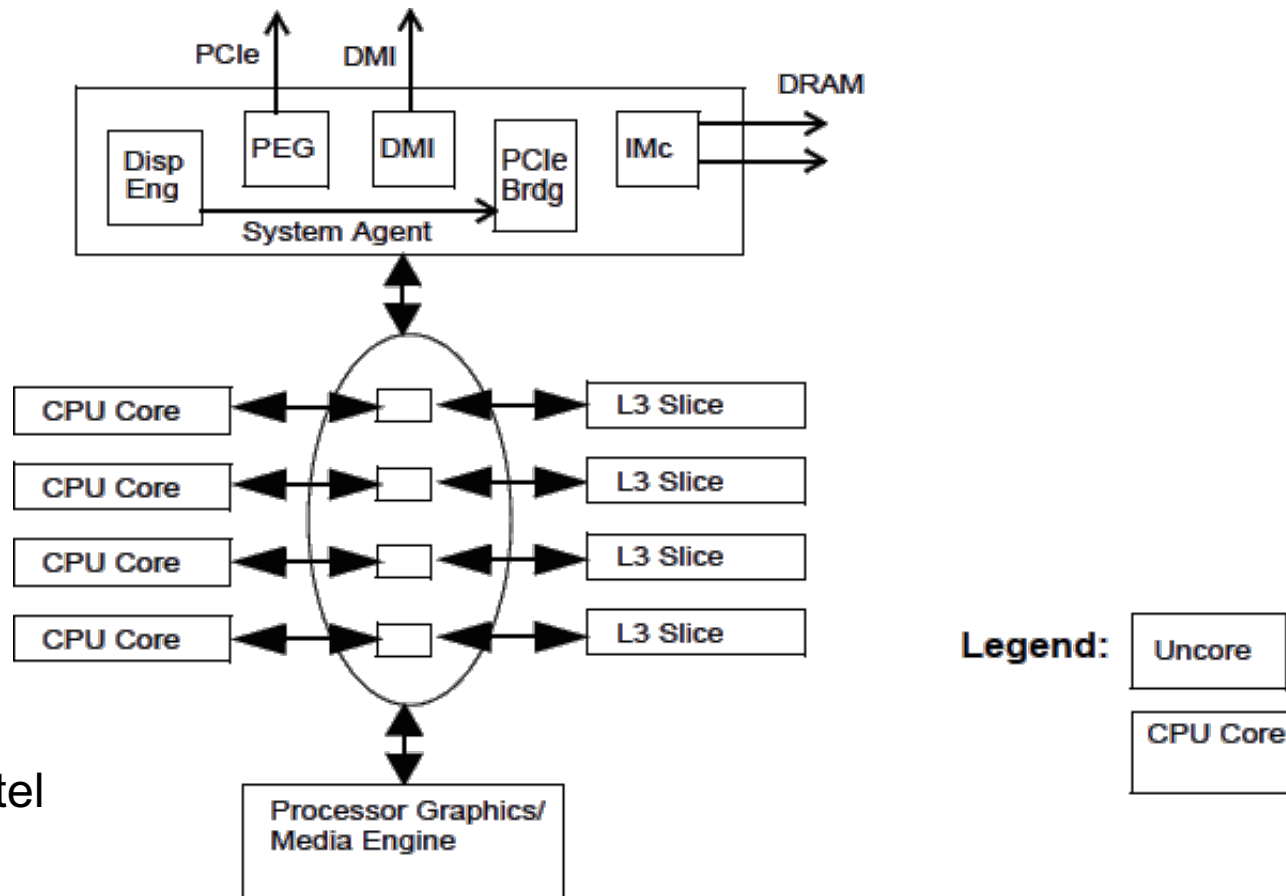
L	I/D	Pages	Assoc	#
1	I	4 KiB–1 GiB?	full?	10
1	D	4 KiB–1 GiB?	full?	10
2	unif	4 KiB–512 MiB	4-way	512

# Intel Core i7 (Haswell) Cache Structure



Source: Intel

# Intel Haswell L3 Cache



Source: Intel

# 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]

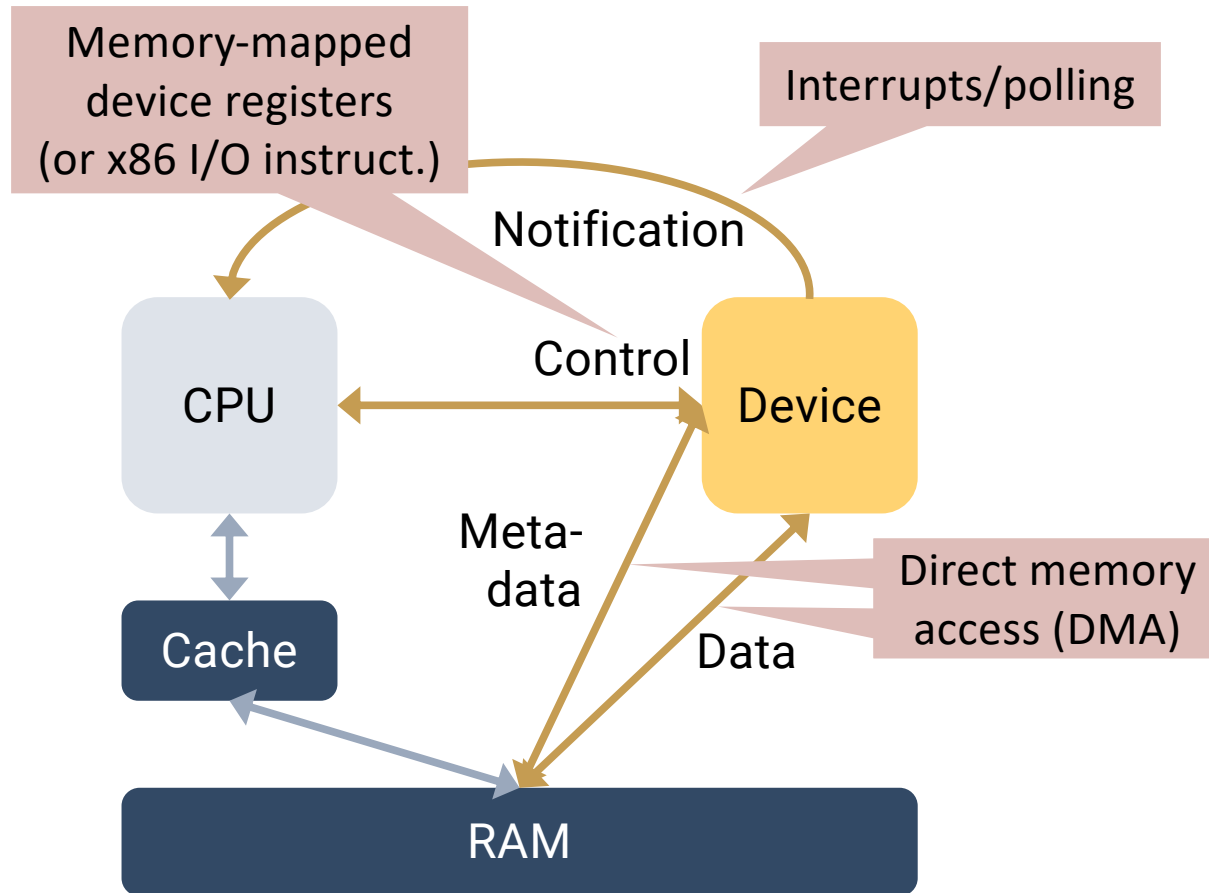
Programmer's model of memory:

```
char *cp, c;  
int32 *ip, i, j, k;  
ip = <addr>;  
cp = (char*)ip;  
j = 0; for (k=0; k<4; k++) j = (j<<8)+*cp++ ;  
i = *ip; // now i==j, assuming big-endian;
```

## Memory contract:

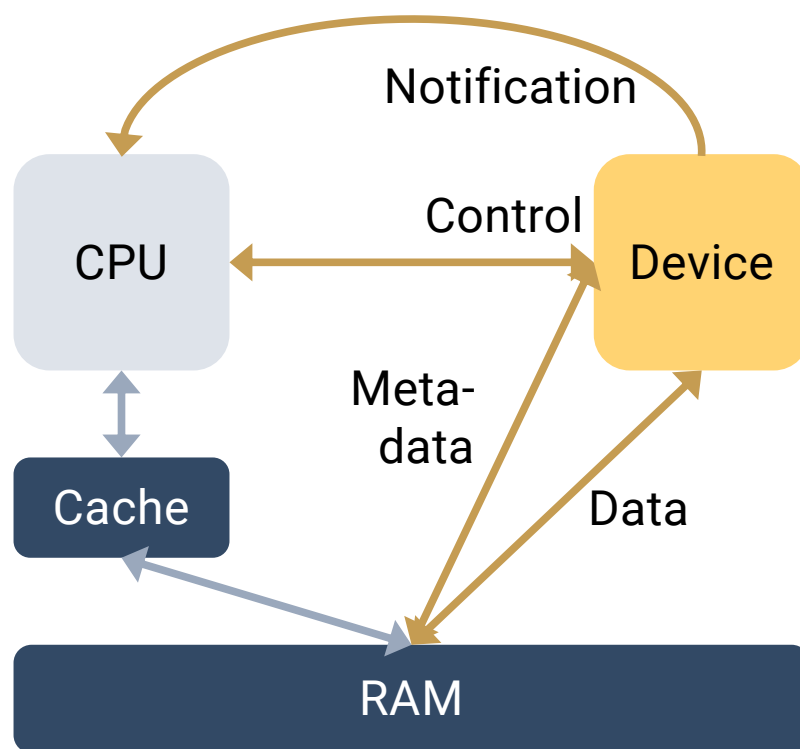
Order or granularity of access don't matter

# Peripheral Devices





# Device-Access Caveats

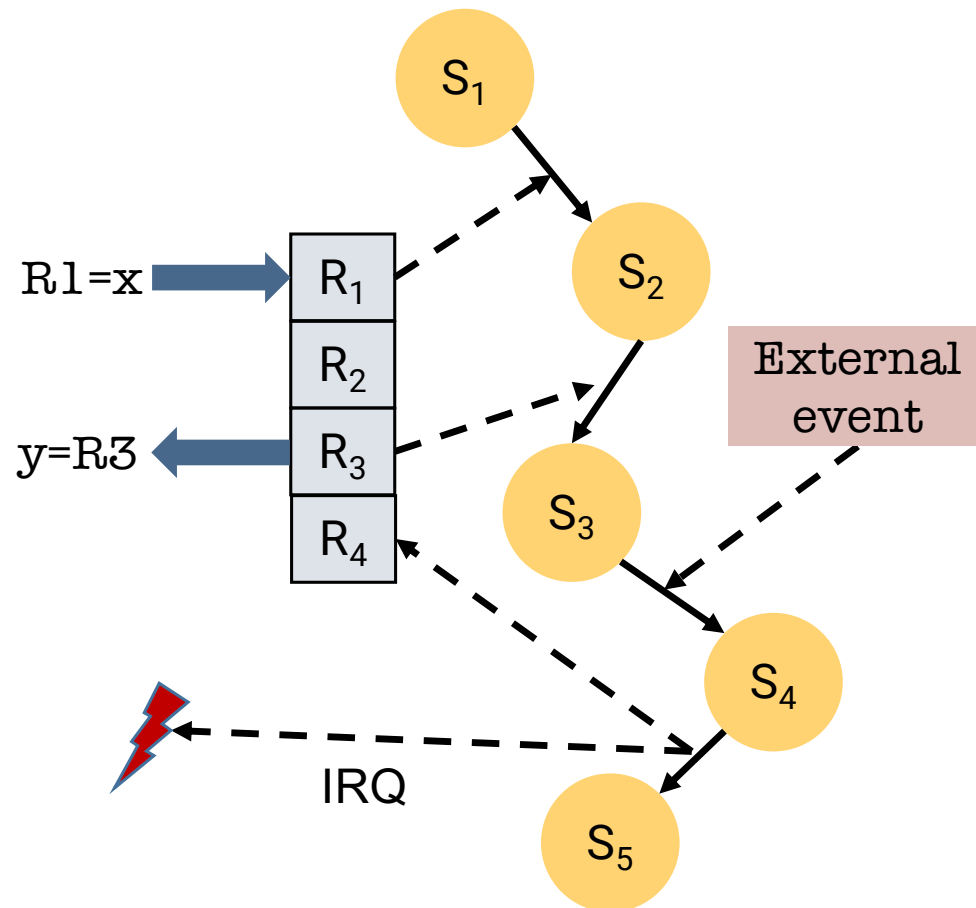


## Device access bypasses cache!

- 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

# Devices Are State Machines



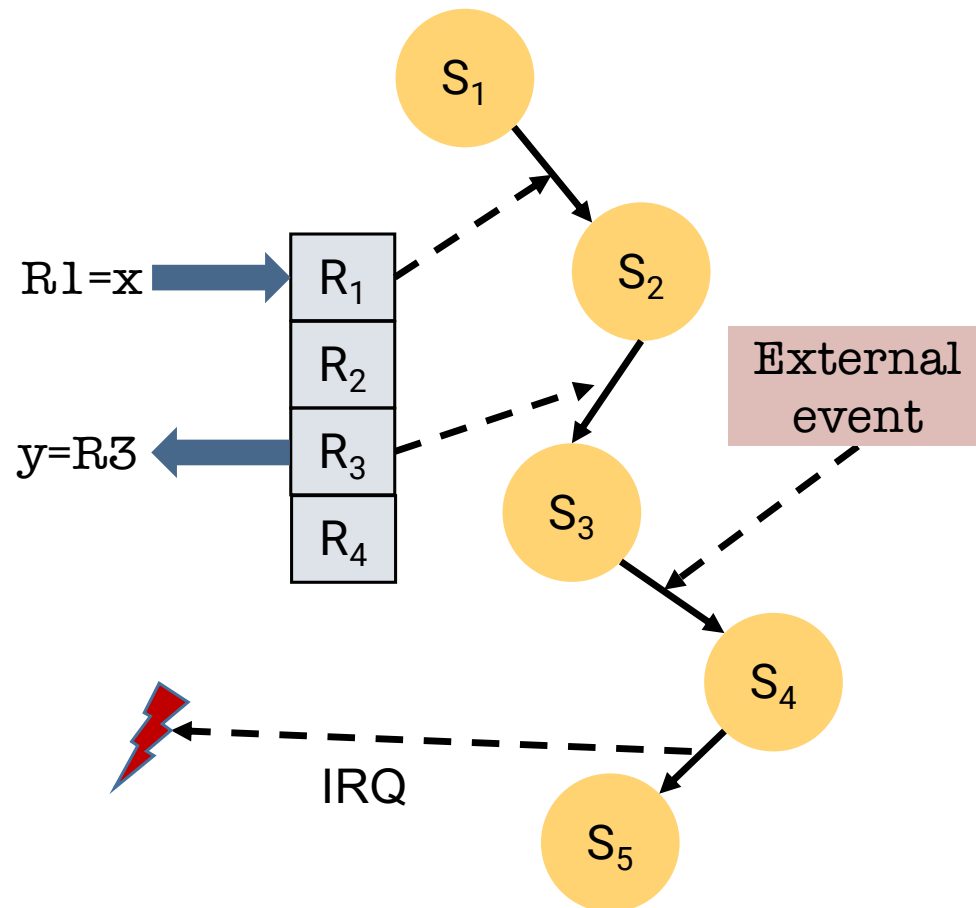
State transitions triggered by:

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

State transitions:

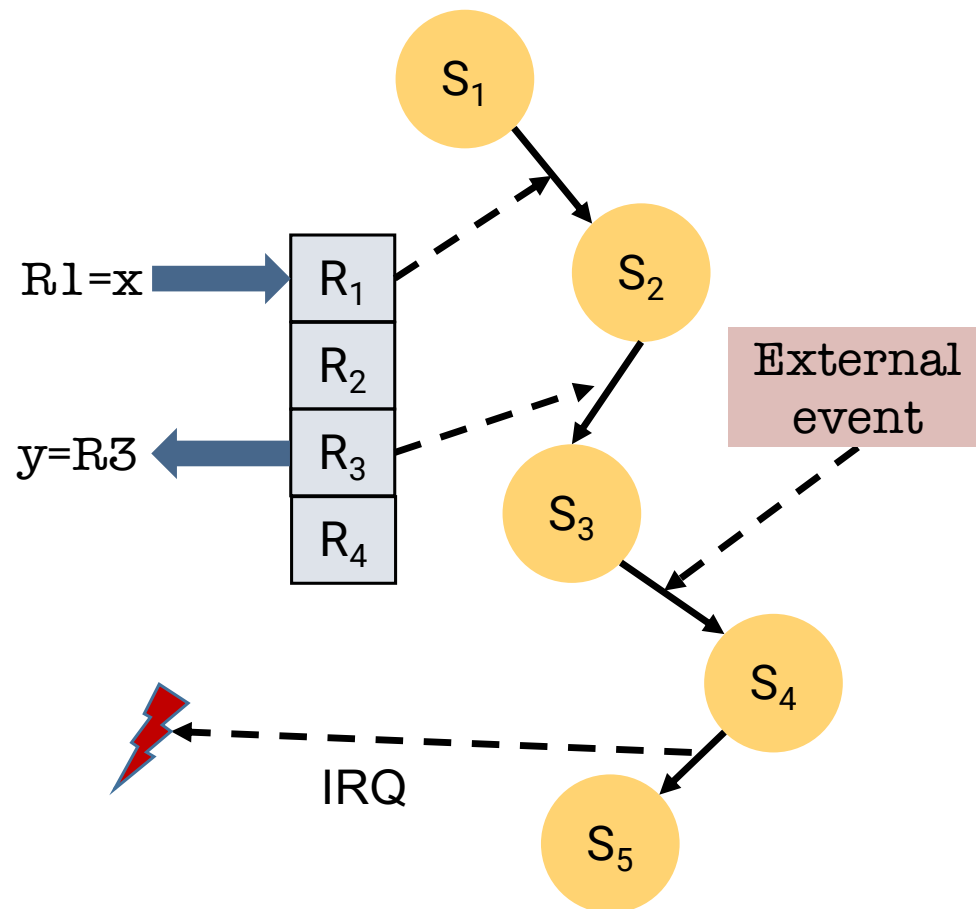
- Change register content
- Raise IRQs

# 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