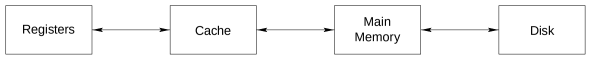
Caching



- 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* ($\geq 90\,\%$)
- Is hardware maintained and (mostly) transparent to software
- Sizes range from few kB to several MB.
- Usually a hierarchy of caches (2–5 levels), on- and off-chip.

Good overview in [Sch94].

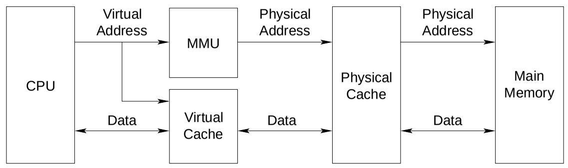
Cache organisation

- Data transfer unit between registers and cache ≤ 1 word (1–8B)
- Date transfer unit between cache and RAM is *cache line*, typically 16–32 bytes, sometimes 128 bytes and more.
- Line is 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.

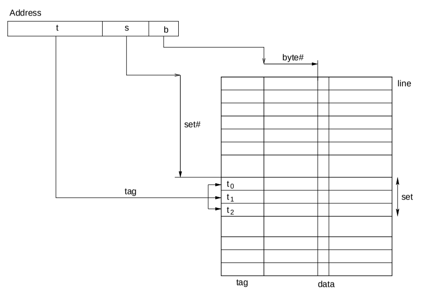
Physically indexed: looked up by *physical address*



Cache access mechanics

- 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 $n = 1 \ldots 5$.
  - ➜ $n = 1$ is called *direct mapped*.
  - ➜ $n = \infty$ is called *fully associative*, unusual for CPU caches.
- Hashing must be simple (complex hardware is slow)
  $\Rightarrow$ use least-significant bits of address.

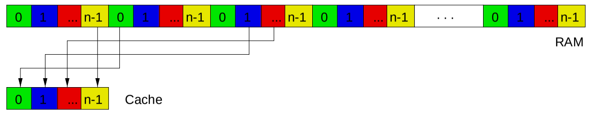
Cache indexing



➜ The *tag* is used to distinguish lines of set...
➜ ... consists of the address bits not used for indexing.

Cache mapping

- Different memory locations map to same cache line:



  - – All memory locations mapping to cache line # i are said to be of *colour* i.
  - – n-way associative cache can hold n lines of the same colour.
    $\Rightarrow$ different types of cache misses:
    - ➜ **capacity misses:** cache is full.
    - ➜ **conflict misses:** set corresponding to address is full, even though there may be unused lines in the cache.

Cache replacement policy

- Indexing (via virtual or physical address) points to line set.
- If all lines of set are valid must replace an existing one.
- Replacement strategy must be simple as it's all done in hardware.
- Dirty bit is used to determine whether a line must be flushed back to memory before invalidation.
- Typical policies:
  - ➜ LRU (or approximation)
  - ➜ FIFO
  - ➜ random

Cache write (update) policy

- Treatment of store operations:
  - – **write back:** Stores update cache only, memory updated when dirty line is replaced (flushed).
    - ➜ Memory is inconsistent with cache.
  - – **write through:** Stores update cache **and** memory immediately.
    - ➜ Memory is always consistent with cache.
- On store to a line not presently in cache, use:
  - – **write allocate:** allocate a cache line to the data and store,
  - – **no allocate:** store to memory and bypass cache.
- Usual combinations: write-back & write-allocate,
  write-through & no-allocate.

Cache types

- **Virtually indexed, virtually tagged** (also called *virtual cache*):
  - ➜ only uses virtual addresses,
  - ➜ can operate concurrently with MMU.
- **Virtually indexed, physically tagged:**
  - ➜ virtual address for accessing line, physical address for tagging,
  - ➜ needs address translation completed for retrieving data,
  - ➜ can index concurrently with MMU, but needs MMU output for tag check.
- **Physically indexed:**
  - ➜ only uses physical addresses
  - ➜ needs address translation completed before begin of access
- Typically, virtually indexed caches are on-chip, physically indexed caches are off-chip.

Virtual Cache Issues

HOMONYMS:

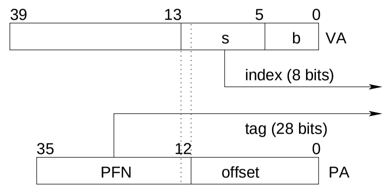
- Same VA corresponds to several PAs
  - ➜ standard situation in multitasking systems (**not** SASOS!)
- Problem: tag may not uniquely identify cache data!
- Homonyms lead to cache accessing the wrong data!
- Homonym prevention:
  - ➜ *tag* virtual address with *address-space ID* (ASID)
    - ➜ disambiguates virtual addresses (makes them globally valid)
  - ➜ use physical tags
  - ➜ flush cache on context switch
  - ➜ use a SASOS

SYNONYMS (ALIASES):

- Several VAs map to the same PA
  - ➜ frames shared between processes
  - ➜ multiple mappings of a frame within an address space
- Synonyms in cache may lead to accessing stale data:
  - ➜ same data may be cached in several lines
  - ➜ on write, one synonym is update
  - ➜ a subsequent read on the other synonym returns the old value
- Physical tagging doesn't help!
- ASIDs don't help either!
- Whether synonyms are a problem depends on cache size and page size!

EXAMPLE: MIPS R4X00 SYNONYMS

- Virtually-indexed, physically tagged on-chip L1 cache
  - ➜ 16kB cache with 32B lines
  - ➜ 4kB (base) page size



- Remember, location of data in cache determined by index
- Tag only confirms whether it's a hit!
- Synonym problem iff $\mathrm{VA}_{12} \neq \mathrm{VA}'_{12}$

SYNONYMS...

- 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 $\mathrm{VA}_{12} = \mathrm{PA}_{12}$

Fully Virtual Caches

- Fastest (don't rely on TLB for retrieving data)
  - ➜ however, needs TLB 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

Fully virtual caches with keys

- Add *address-space identifier* (ASID) as part of tag.
- On access compare with CPU's ASID register.
- Turns homonyms into synonyms
  - ➜ Removes homonym problem
  - ➜ Potentially better context switching performance
  - ➜ ASID recycling still requires cache flush
- Doesn't solve synonym problem
- Doesn't solve write-back problem

Virtually indexed, physically tagged caches

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

Physical Caches

- 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.
  - ➜ Attractive feature for intermediate level cache.
- Cache can use *bus snooping* to receive/supply DMA data
- Usable as off-chip cache with any architecture.

Cache Hierarchy

- Use a hierarchy of caches to balance memory accesses:
  - ➜ Small, fast, virtually indexed cache on top (L1 cache).
  - ➜ Large, slow, physically indexed cache at bottom (L2–L5).
- Each level reduces and clusters traffic.
- High levels tend to be separated into instruction and data caches.
- Low levels tend to be unified.

Translation Lookaside Buffers

- TLB is a cache for page table entries.
- Can be:
  - ➜ hardware loaded, transparent to OS, or
  - ➜ software loaded, maintained by OS.
- Can be:
  - ➜ split, instruction and data TLBs, or
  - ➜ unified.
- Some architectures (MIPS, IA-64) use a hierarchy of TLBs:
  - ➜ Top-level TLB is hardware-loaded from lower levels.
  - ➜ Transparent to OS.

TLB Issues

Associativity

- First TLB (VAX-11/780, [CE85]) was 2-way associative.
- Most modern architectures have fully associative TLBs.
- Exceptions:
  - ➜ i486 (4-way),
  - ➜ Pentium, Pentium-6 (4-way),
  - ➜ IBM RS/6000 (2-way).
- Superpages ⇒ fully associative TLB

Size (I-TLB + D-TLB)

Architecture	*TLB Size*
VAX	64–256
ix86	32–32+64
MIPS	96–128
SPARC	64
Alpha	32–128+128
RS/6000	32+128
PA-8000	96+96
Itanium	64+96

Note: not much growth in 20 years!

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.

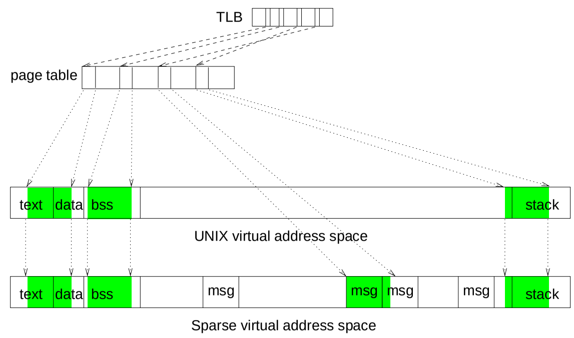
Address Space Usage vs. TLB Coverage

- Each TLB entry maps one virtual page.
- On TLB miss, reloaded from page table (PT), which is in memory.

⇒ Some TLB entries need to map page table.
  - ➜ E.g. 32-bit page table entries, 4kb pages.
  - ➜ One PT page maps 4Mb.

- Traditional UNIX process has 2 regions of allocated virtual address space:
  - ➜ low end: text, data, heap,
  - ➜ high end: stack.
  - ⇒ 2–3 PT pages are sufficient to map most address spaces.

Sparse address space use ties up PT entries



Origins of sparse address-space use

- Modern OS features:
  - ➜ memory-mapped files,
  - ➜ dynamically-linked libraries,
  - ➜ mapping IPC (server-based systems)...
- This problem gets worse 64-bit address spaces:
  - ➜ bigger page tables.
- An in-depth study of such effects can be found in [UNS+94].

References

[CE85] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *Trans. Comp. Syst.*, 3:31–62, 1985.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.

[UNS+94] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest, and Richard Brown. Design tradeoffs for software-managed TLBs. *Trans. Comp. Syst.*, pages 175–205, 1994.