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].

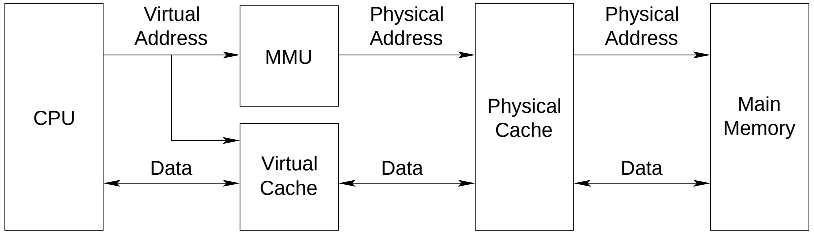
CACHE ORGANISATION

- Unit of data transfer between registers and cache is ≤ 1 word (1–8B)
- Unit of data transfer 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: virtual address is used to look up entry operates concurrently with address translation.

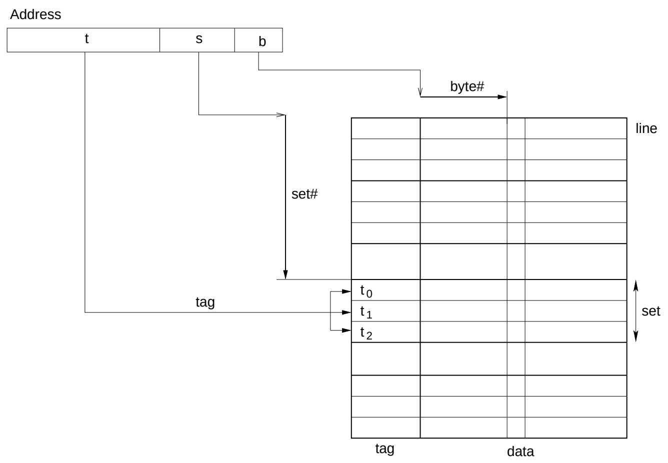
Physically indexed: physical address is used to look up entry.



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 caches.
- Hashing must be simple (complex hardware is slow) $\Rightarrow$ use least-significant bits of address.

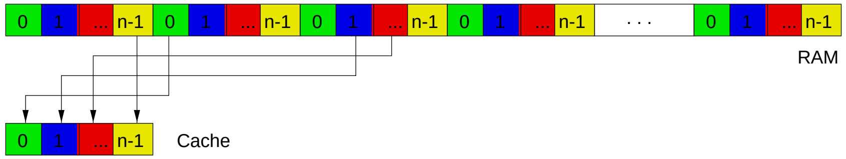
CACHE INDEXING:



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

CACHE MAPPING

Many different memory locations map to the 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 Caches

- Fastest.
- Aliasing problems:
  - ➜ Two processes using same virtual address for different physical locations would access other process' data after context switch.
  - ➜ Must flush cache on each context switch (unless guarantee no overlap).
    ⇒ Context switches are expensive.
  - ➜ May also have to flush when leaving kernel mode to prevent user code from accessing kernel data.

Used on MC68040, i860

Virtual caches with keys

- Add *address-space identifier* (ASID) as part of tag.
- On access compare with CPU's ASID register.
- Removes aliasing (& kernel data) problems.
- Context switches fast.
- ASID recycling requires cache flush.

Virtually indexed, physically tagged caches

- No aliasing problem
- Easy to manage
- More bits per tag (cannot leave off set-number bits).
- Incoherencies when sharing memory via different virtual addresses.

Used on most modern architectures for L1 cache.

Physical caches

- No aliasing 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 watching* to receive/supply DMA data

Usable as off-chip cache with any architecture.

Cache hierarchy

- Most systems 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 cache).
- 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).

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*.
- In addition, many modern architectures have software-loaded TLBs.

⇒ General increase in TLB miss handling cost.

Address Space Usage vs. TLB Coverage

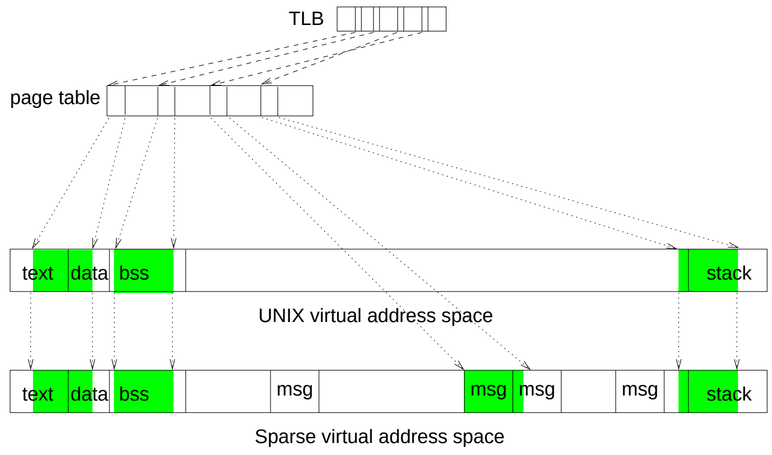
- Each TLB entry maps one virtual page.
- On TLB miss, reloaded from page table (PT), which is in memory.

⇒ 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

- ➜ 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. *ACM 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. *ACM Trans. Comp. Syst.*, pages 175–205, 1994.