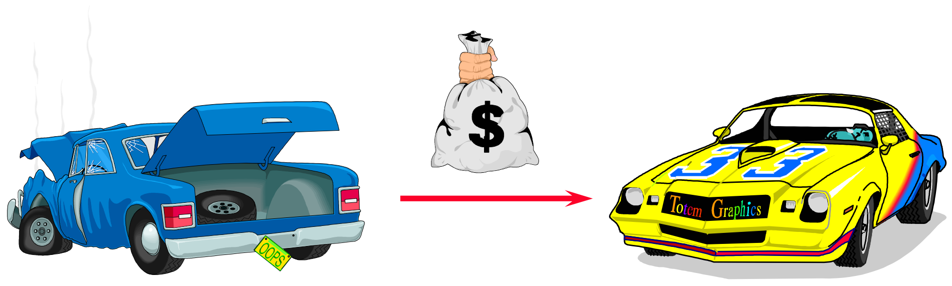
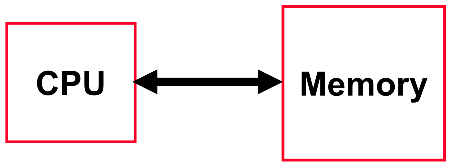
Multilevel Memories (Improving performance using a little "cash")



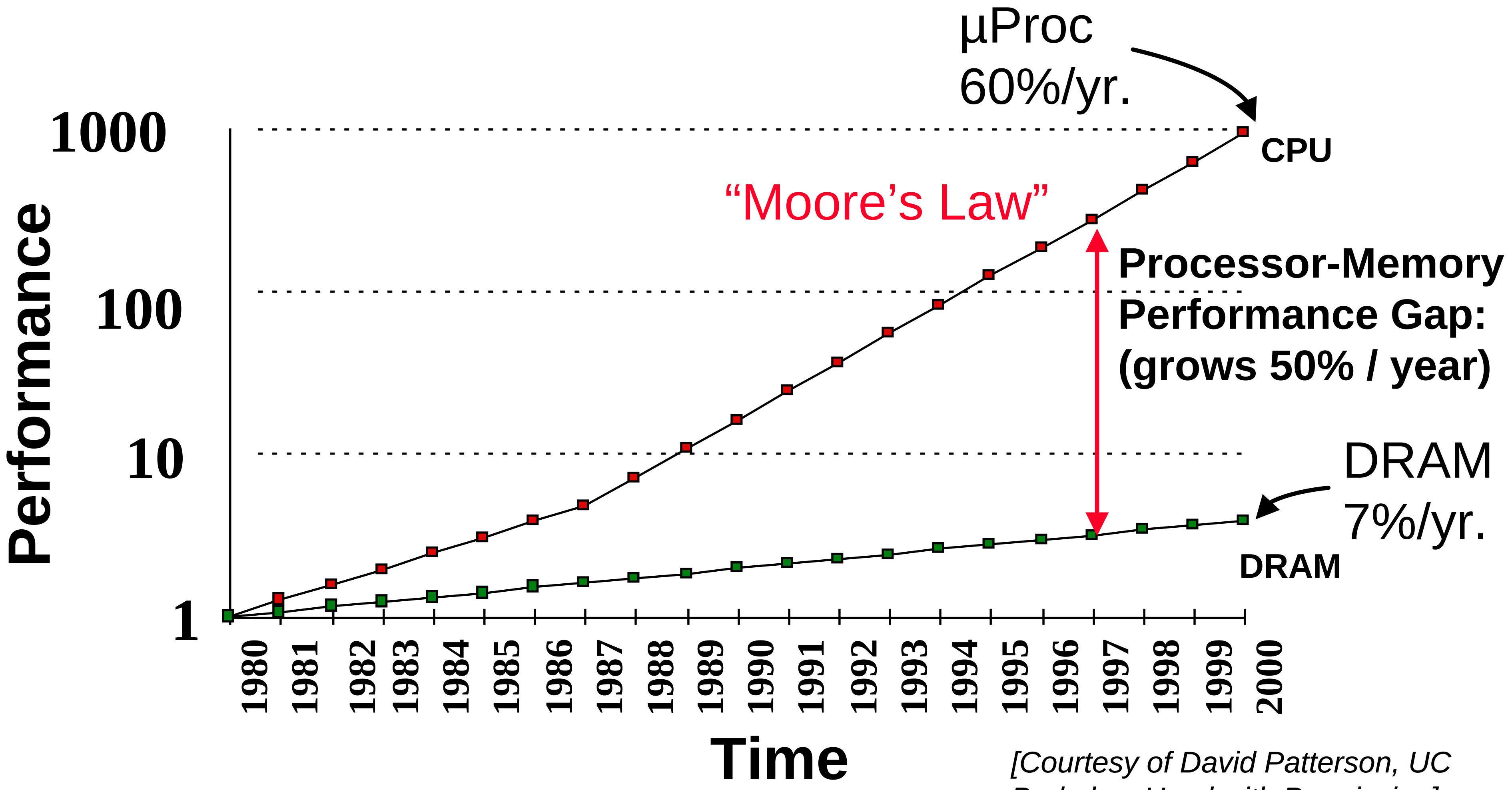
CPU-Memory Bottleneck



Performance of high-speed computers is usually limited by memory *bandwidth* & *latency*

- ***Latency (time for a single access)***
  Memory access time >> Processor cycle time
- ***Bandwidth (number of accesses per unit time)***
  if fraction *m* of instructions access memory,
  $\Rightarrow$ 1+*m* memory references / instruction
  $\Rightarrow$ CPI = 1 requires 1+*m* memory refs / cycle

Processor-DRAM Gap (latency)



[Courtesy of David Patterson, UC Berkeley; Used with Permission]

Four-issue superscalar executes 400 instructions during cache miss!

Multilevel Memory

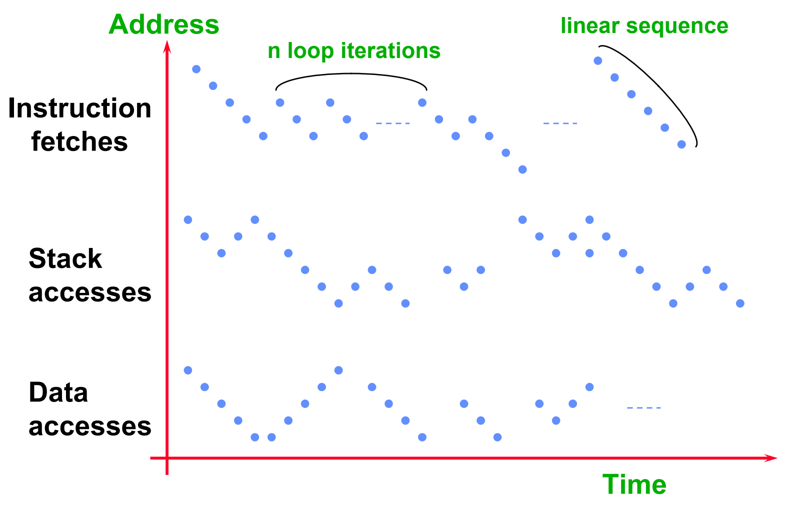
Strategy: <u>Hide</u> latency using small, fast memories called caches.

Caches are a mechanism to hide memory latency based on the empirical observation that the stream of memory references made by a processor exhibits locality

```
                                   PC
          …                        96
loop: ADD r2, r1, r1              100
      SUBI r3, r3, #1             104
      BNEZ r3, loop               108
          …                       112
```

What is the pattern of instruction memory addresses?

Typical Memory Reference Patterns



5

Locality Properties of Patterns

Two locality properties of memory references:

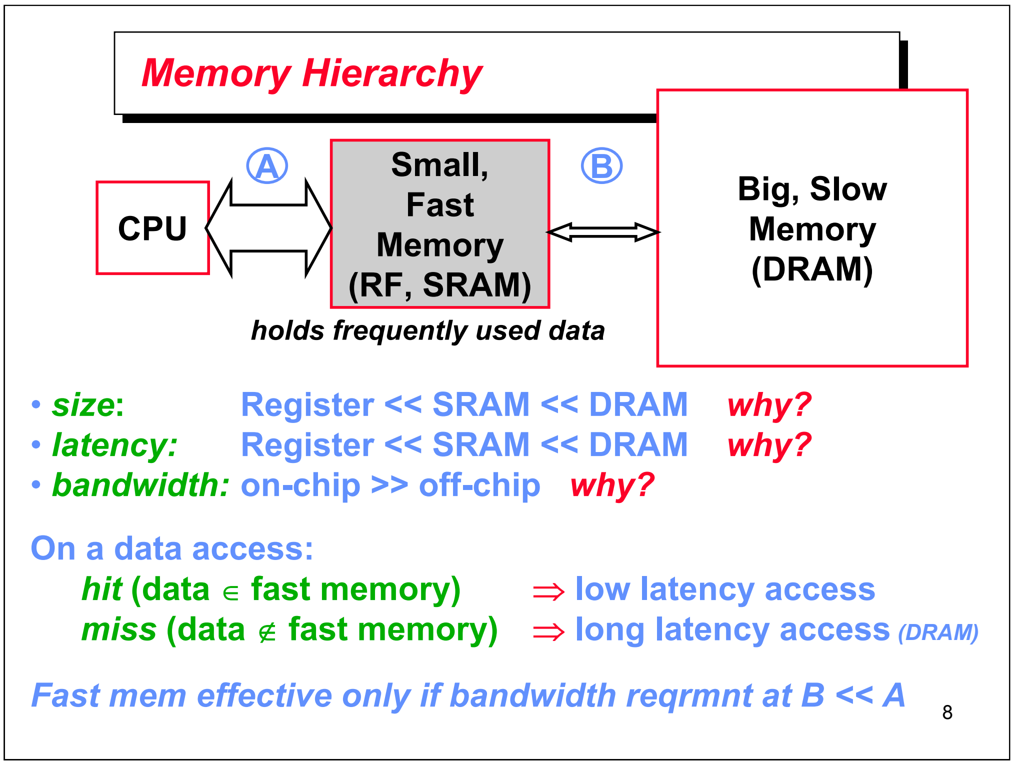
- If a location is referenced it is likely to be referenced again in the near future.
  This is called temporal locality.
- If a location is referenced it is likely that locations near it will be referenced in the near future.
  This is called spatial locality.

Caches

Caches exploit both properties of patterns.

- **Exploit temporal locality by remembering the contents of recently accessed locations.**

- **Exploit spatial locality by remembering blocks of contents of recently accessed locations.**



Memory Hierarchy

CPU ⇔ (A) Small, Fast Memory (RF, SRAM) ⇔ (B) Big, Slow Memory (DRAM)

holds frequently used data

- ***size***: Register << SRAM << DRAM ***why?***
- ***latency:*** Register << SRAM << DRAM ***why?***
- ***bandwidth:*** on-chip >> off-chip ***why?***

On a data access:

hit (data $\in$ fast memory) $\Rightarrow$ low latency access

miss (data $\notin$ fast memory) $\Rightarrow$ long latency access *(DRAM)*

Fast mem effective only if bandwidth reqrmnt at B << A

8

Due to cost

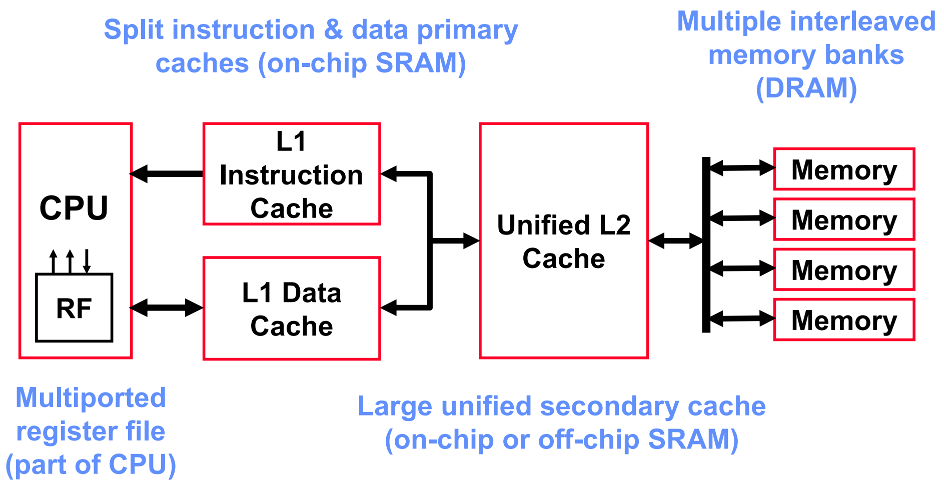
Due to size of DRAM

Due to cost and wire delays (wires on-chip cost much less, and are faster)

Management of Memory Hierarchy

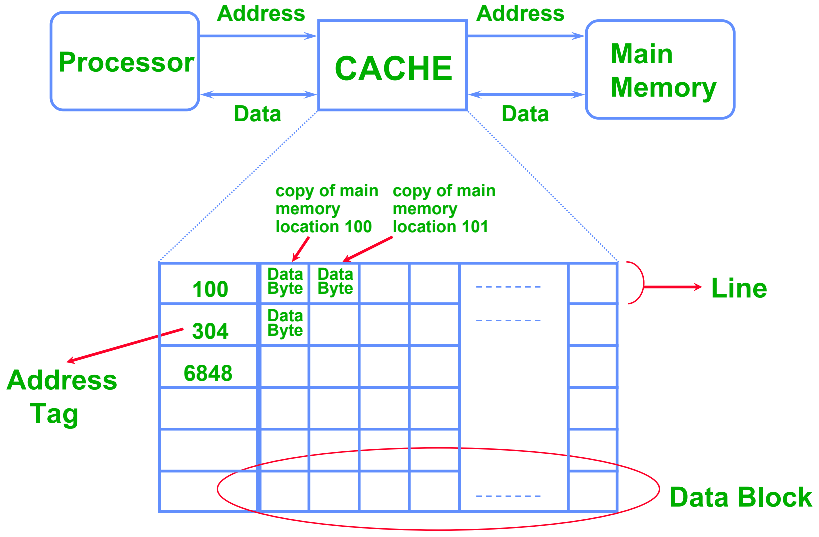
- **Software managed, e.g., registers**
  - **part of the software-visible processor state**
  - **software in complete control of storage allocation**
    - » **but hardware might do things behind software's back, e.g., register renaming**
- **Hardware managed, e.g., caches**
  - **not part of the software-visible processor state**
  - **hardware automatically decides what is kept in fast memory**
    - » **but software may provide "hints", e.g., don't cache or prefetch**

A Typical Memory Hierarchy c.2000



Split instruction & data primary caches (on-chip SRAM)
Multiple interleaved memory banks (DRAM)
CPU
RF
L1 Instruction Cache
L1 Data Cache
Unified L2 Cache
Memory
Memory
Memory
Memory
Multiported register file (part of CPU)
Large unified secondary cache (on-chip or off-chip SRAM)

Inside a Cache



Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either

Found in cache
a.k.a. HIT

Not in cache
a.k.a. MISS

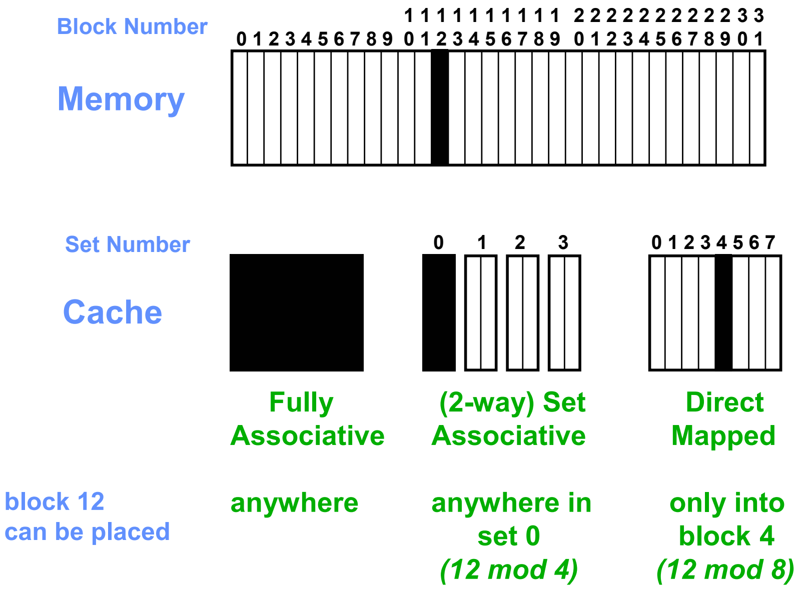
Return copy of data from cache

Read block of data from Main Memory

Wait …

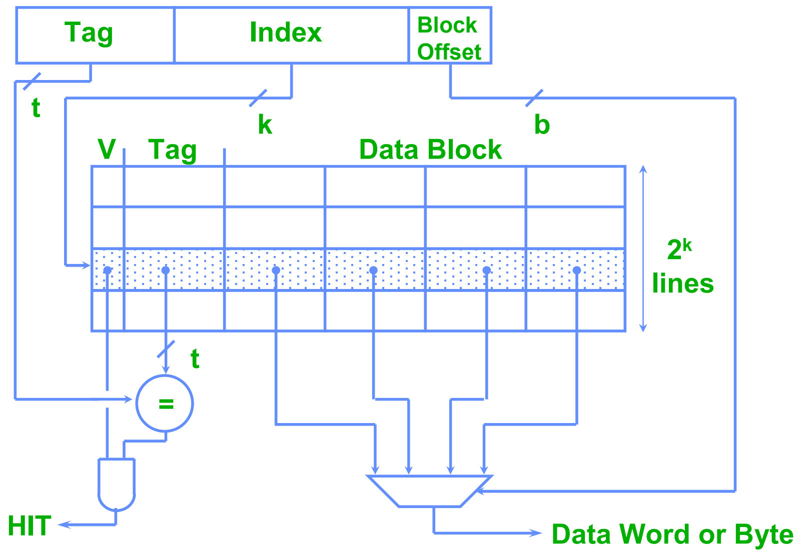
Which line do we replace? → **Return data to processor and update cache**

Placement Policy

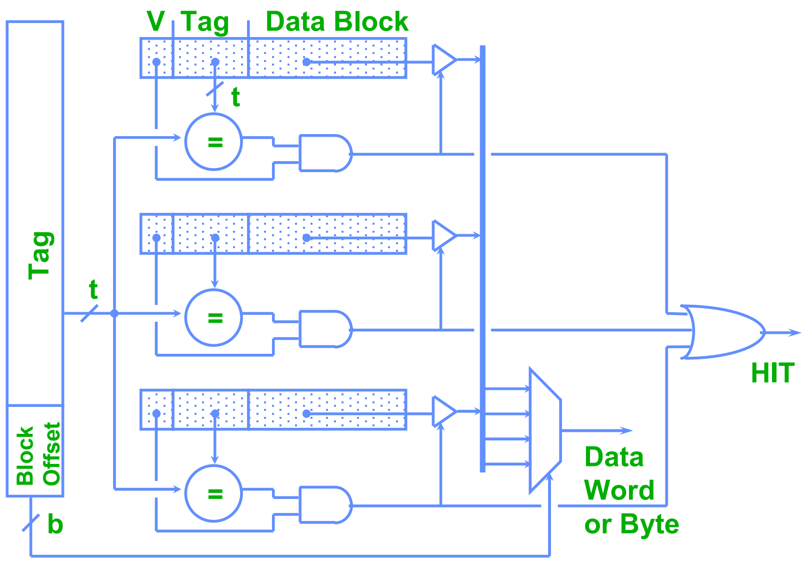


13

Direct-Mapped Cache



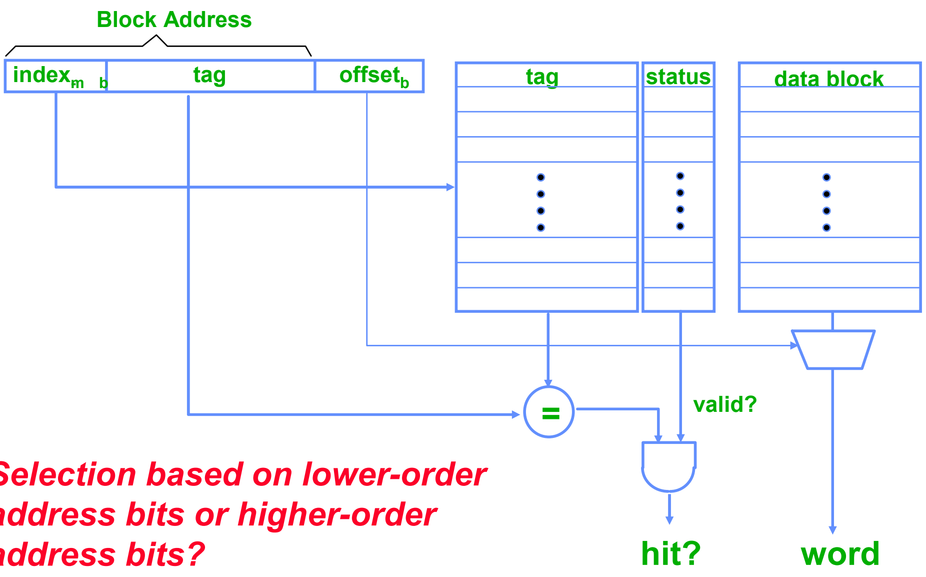
Fully Associative Cache



15

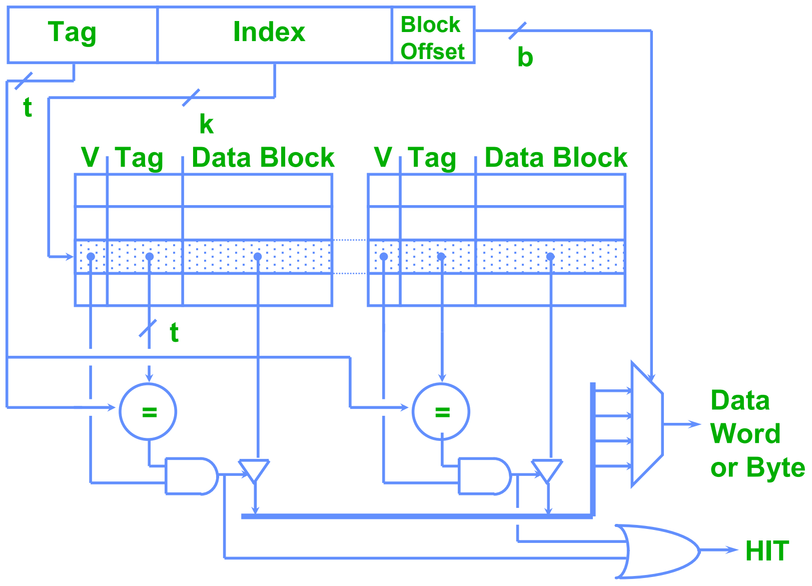
Direct Map Address Selection

higher-order vs. lower-order address bits



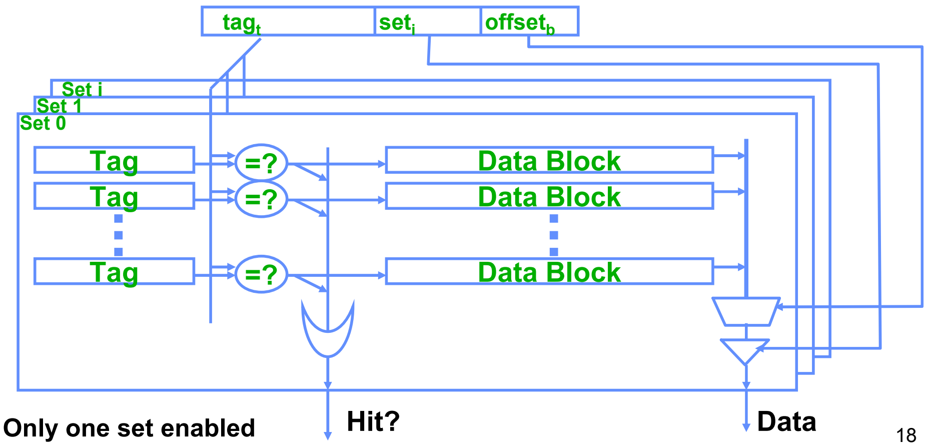
Selection based on lower-order address bits or higher-order address bits?

2-Way Set-Associative Cache



Highly-Associative Caches

- **For high associativity, use content-addressable memory (CAM) for tags**
- ***Overhead?***



Only one set enabled
Only hit data accessed

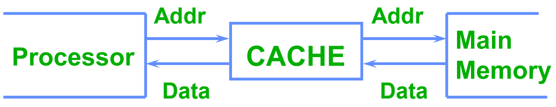
Advantage is low power because only hit data is accessed.

Average Read Latency using a Cache

α is HIT RATIO: Fraction of references in cache

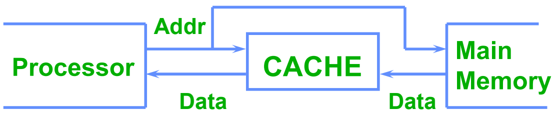
$1 - \alpha$ is MISS RATIO: Remaining references

Average access time for serial search:



$t_c + (1 - \alpha)\, t_m$

Average access time for parallel search:



$\alpha\, t_c + (1 - \alpha)\, t_m$

t_c *is smallest for which type of cache?*

Write Policy

Cache hit:

write through*:* write both cache & memory

- generally higher traffic but simplifies cache coherence

write back*:* write cache only (memory is written only when the entry is evicted)

- a dirty bit per block can further reduce the traffic

Cache miss:

no write allocate*:* only write to main memory

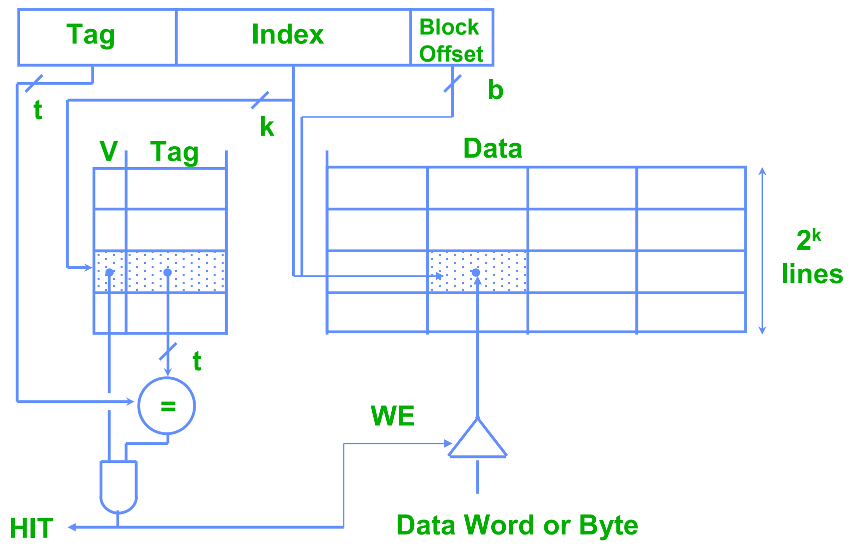
write allocate*:* *(aka fetch on write)* fetch block into cache

Common combinations:

write through and no write allocate

write back with write allocate

Write Performance



Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

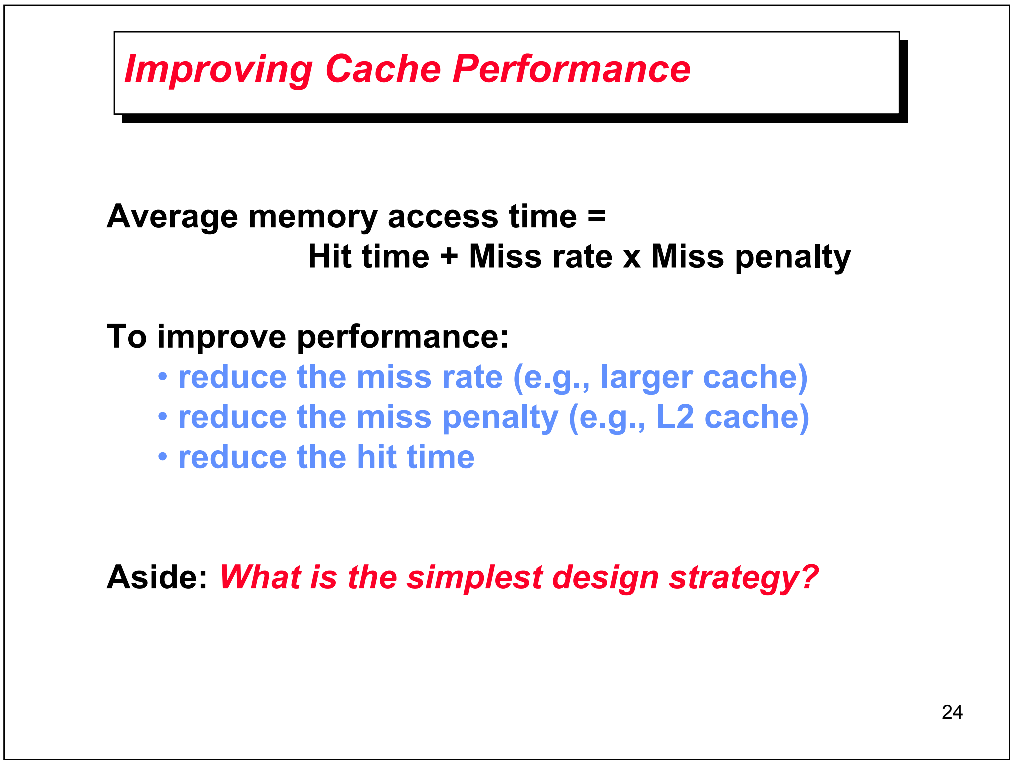
- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way easy)
  - pseudo-LRU binary tree often used for 4-8 way
- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches

This is a second-order effect. Why?

Causes for Cache Misses

- ***Compulsory:*** first-reference to a block *a.k.a.* cold start misses
  - misses that would occur even with infinite cache
- ***Capacity:*** cache is too small to hold all data needed by the program
  - misses that would occur even under perfect placement & replacement policy
- ***Conflict:*** misses that occur because of collisions due to block-placement strategy
  - misses that would not occur with full associativity

Determining the type of a miss requires running program traces on a cache simulator



Improving Cache Performance

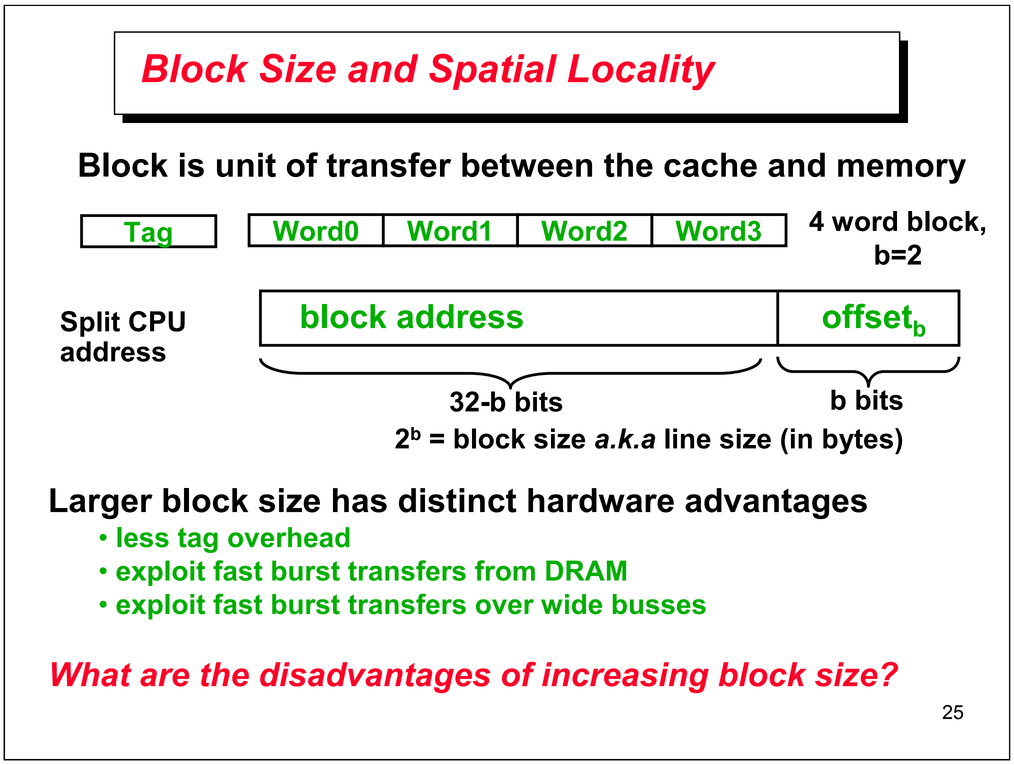
Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:

- **reduce the miss rate (e.g., larger cache)**
- **reduce the miss penalty (e.g., L2 cache)**
- **reduce the hit time**

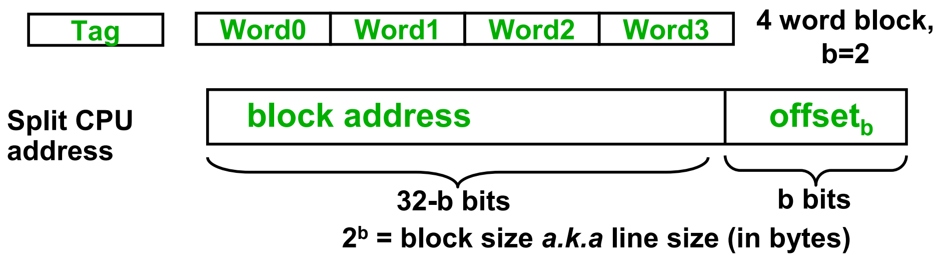
Aside: ***What is the simplest design strategy?***

Design the largest primary cache without slowing down the clock
Or adding pipeline stages.



Block Size and Spatial Locality

Block is unit of transfer between the cache and memory



2^b = block size *a.k.a* line size (in bytes)

Larger block size has distinct hardware advantages

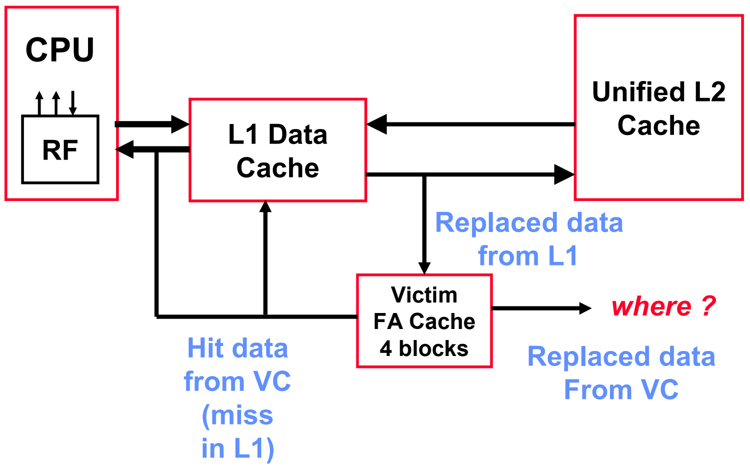
- **less tag overhead**
- **exploit fast burst transfers from DRAM**
- **exploit fast burst transfers over wide busses**

What are the disadvantages of increasing block size?

25

Larger block size will reduce compulsory misses (first miss to a block). Larger blocks may increase conflict misses since the number of blocks is smaller.

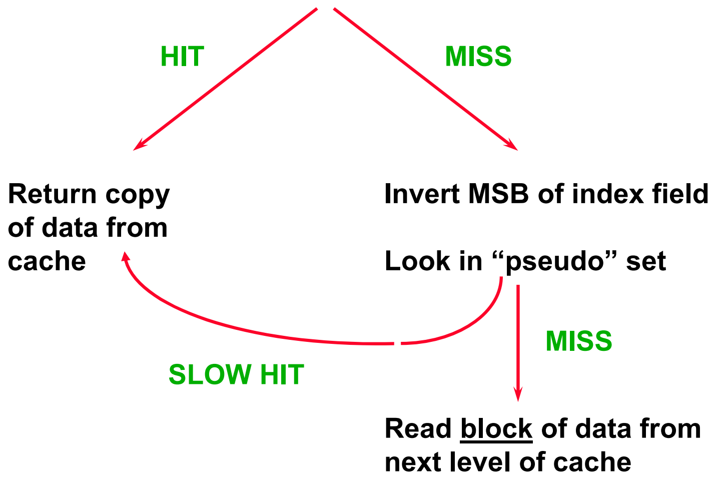
Victim Caches (HP 7200)

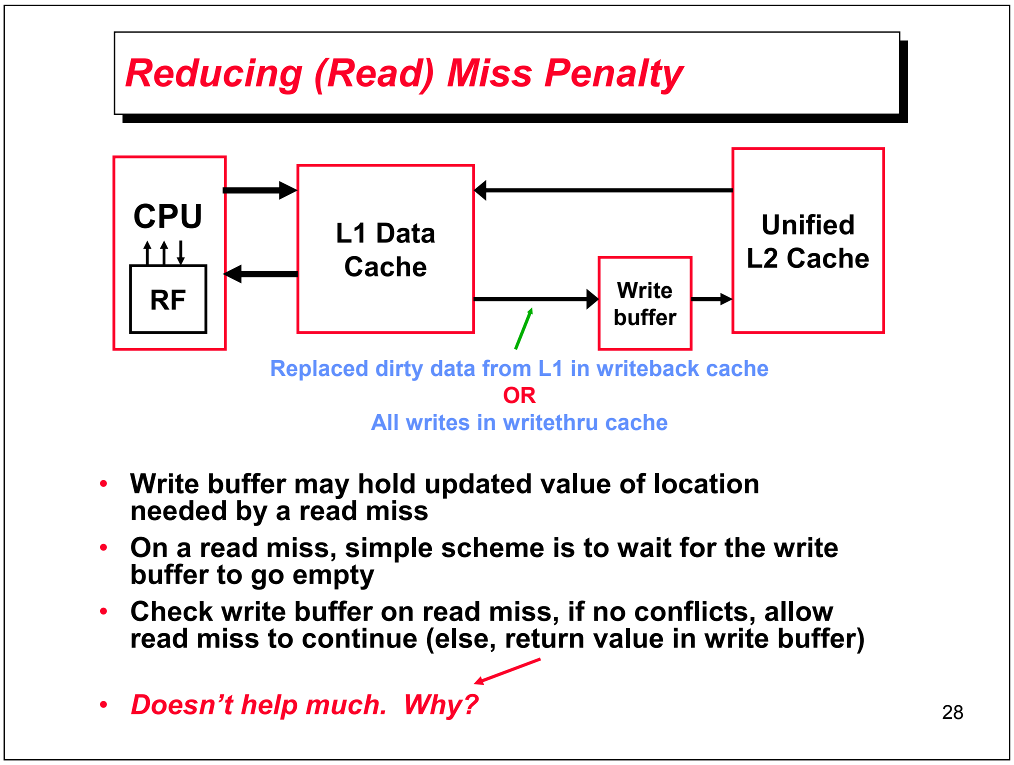


Works very well with a direct-mapped cache

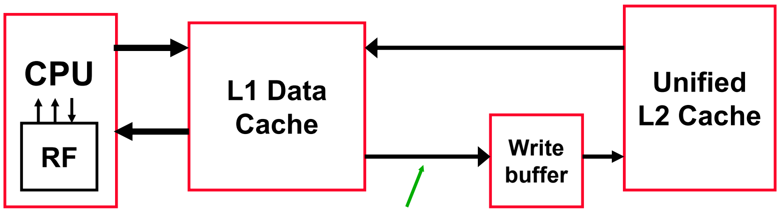
Pseudo-Associative Caches (MIPS R1000)

Look at Processor Address, look in indexed set for data. Then either


HIT
MISS
Return copy of data from cache
Invert MSB of index field
Look in "pseudo" set
MISS
SLOW HIT
Read block of data from next level of cache



Reducing (Read) Miss Penalty



Replaced dirty data from L1 in writeback cache
OR
All writes in writethru cache

- Write buffer may hold updated value of location needed by a read miss
- On a read miss, simple scheme is to wait for the write buffer to go empty
- Check write buffer on read miss, if no conflicts, allow read miss to continue (else, return value in write buffer)

- *Doesn't help much. Why?*

28

Deisgners of the MIPS M/1000 estimated that waiting for a four-word buffer to empty

increased the read miss penalty by a factor of 1.5.

Block-level Optimizations

- Tags are too large, i.e., too much overhead
  - Simple solution: Larger blocks, but miss penalty could be large.
- Sub-block placement
  - A valid bit added to units smaller than the full block, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the word in the cache?*

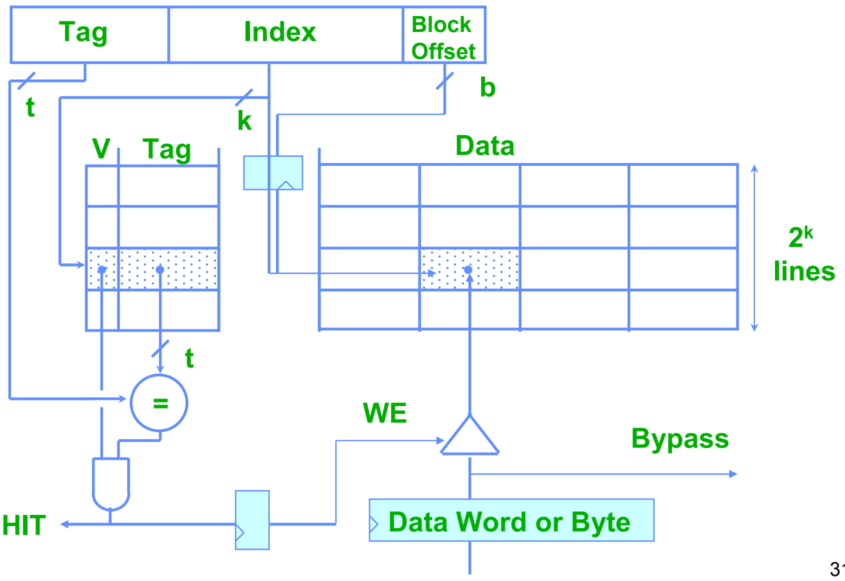
100
300
204

1		*1*		*1*		*1*	
1		*1*		*0*		*0*	
0		*1*		*0*		*1*	

Write Alternatives

- Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit
- Design data RAM that can perform read *and* write in one cycle, restore old value after tag miss
- Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check
  - Need to bypass from write buffer if read matches write buffer tag

Pipelining Cache Writes (Alpha 21064)



Effect of Cache Parameters on Perf.

- **Larger cache size**
  - + reduces capacity and conflict misses
  - - hit time may increase
- **Larger block size**
  - + spatial locality reduces compulsory misses and capacity reload misses
  - - fewer blocks may increase conflict miss rate
  - - larger blocks may increase miss penalty
- **Higher associativity**
  - + reduces conflict misses (up to around 4-8 way)
  - - may increase access time

➢ **See page 427 of text for a nice summary**