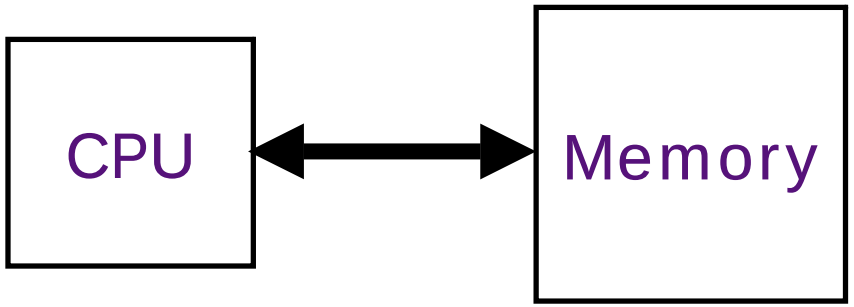
Cache Organization

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

Multilevel Memory

Strategy: Reduce average latency using small, fast memories called caches.

Caches are a mechanism to reduce memory latency based on the empirical observation that the patterns of memory references made by a processor are often highly predictable:

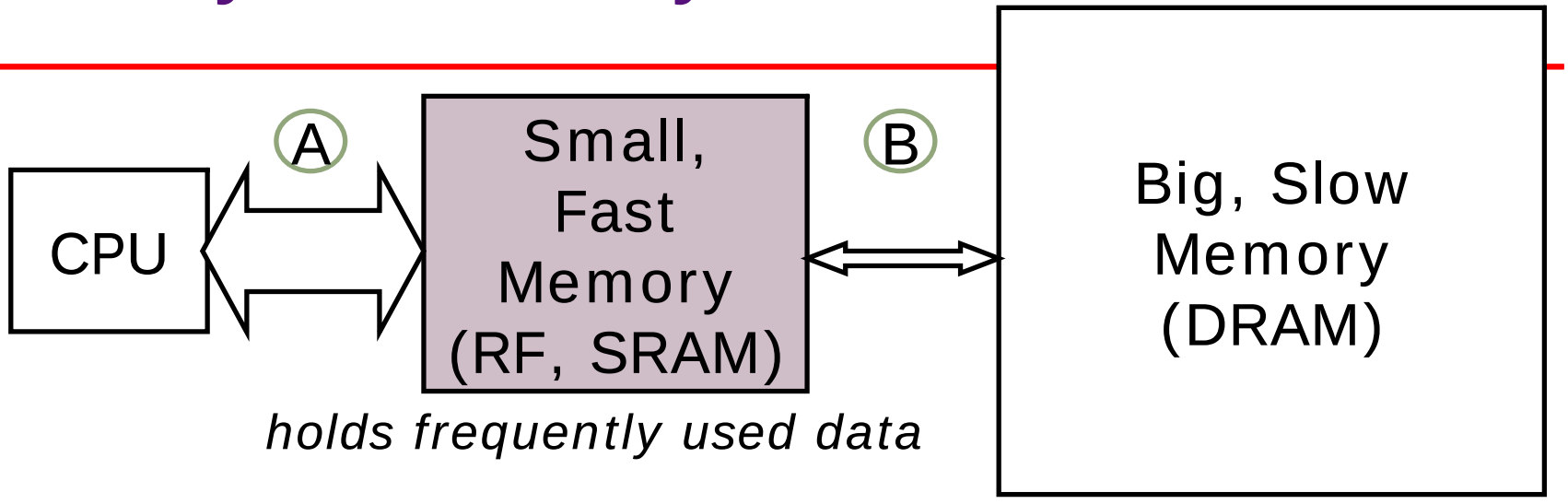
```
                                    PC
            ...                     96
loop: ADD r2, r1, r1                100
      SUBI r3, r3, #1               104
      BNEZ r3, loop                 108
            ...                     112
```

Common Predictable Patterns

Two predictable properties of memory references:

- *Temporal Locality:* If a location is referenced it is likely to be referenced again in the near future.
- *Spatial Locality:* If a location is referenced it is likely that locations near it will be referenced in the near future.

Memory Hierarchy



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

On a data access:

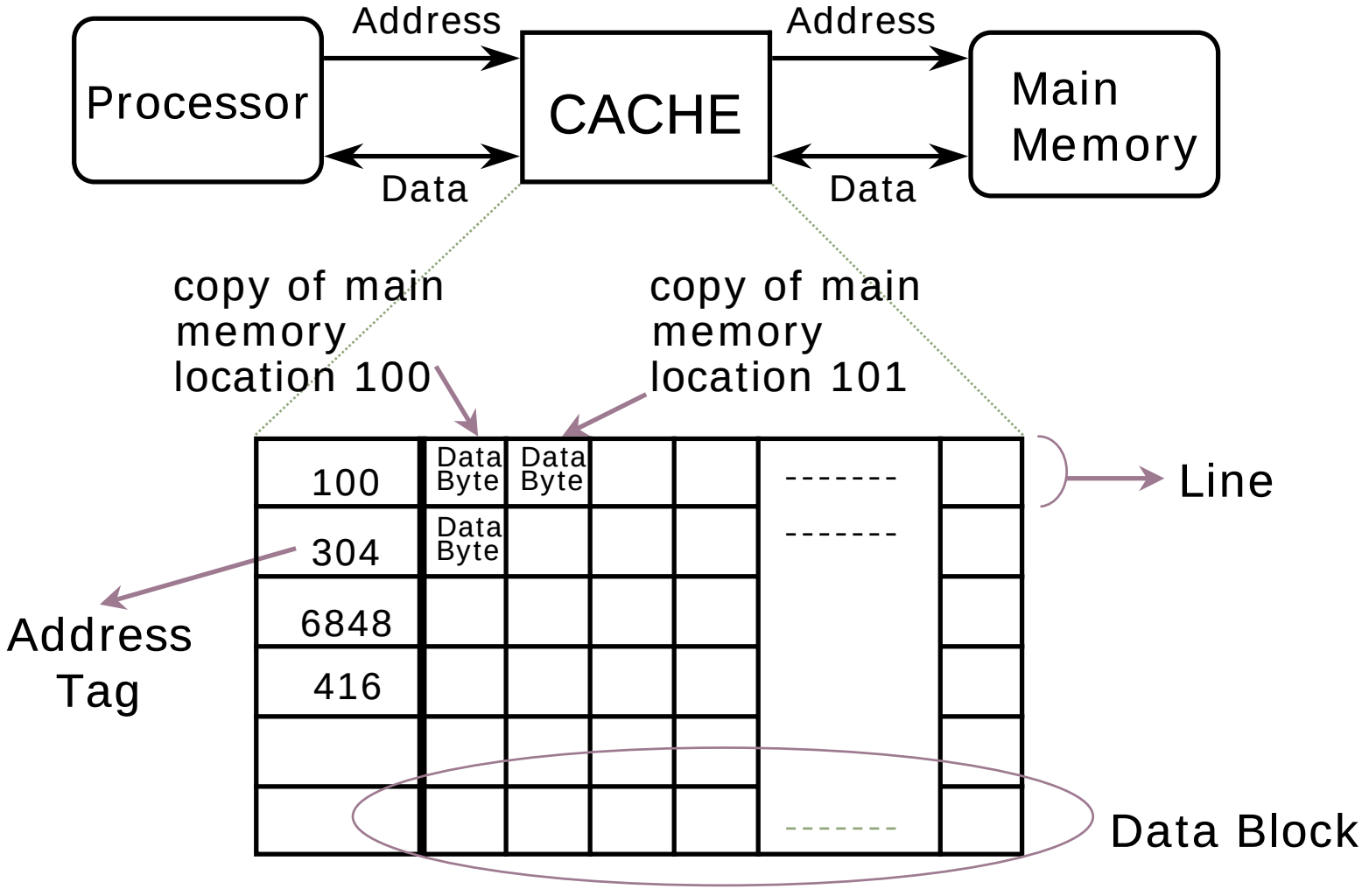
hit (data ∈ fast memory) ⇒ low latency access

miss (data ∉ fast memory) ⇒ long latency access *(DRAM)*

Management of Memory Hierarchy

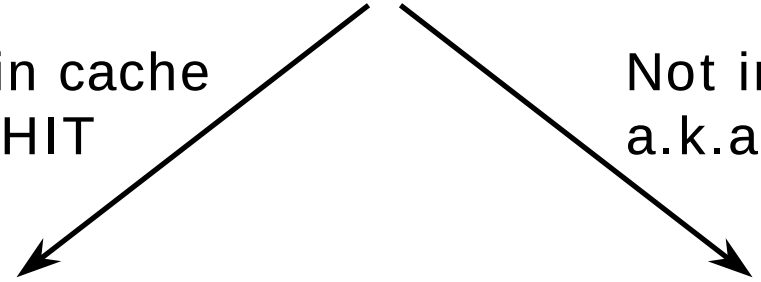
- Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    - but hardware might do things behind software's back, e.g., stack management, register renaming
- Large/slower storage, e.g., memory
  - Address usually computed from values in register
  - Generally implemented as a cache hierarchy
    - hardware decides what is kept in fast memory
    - but software may provide "hints", e.g., don't cache or prefetch

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

Return copy of data from cache

Not in cache
a.k.a. MISS

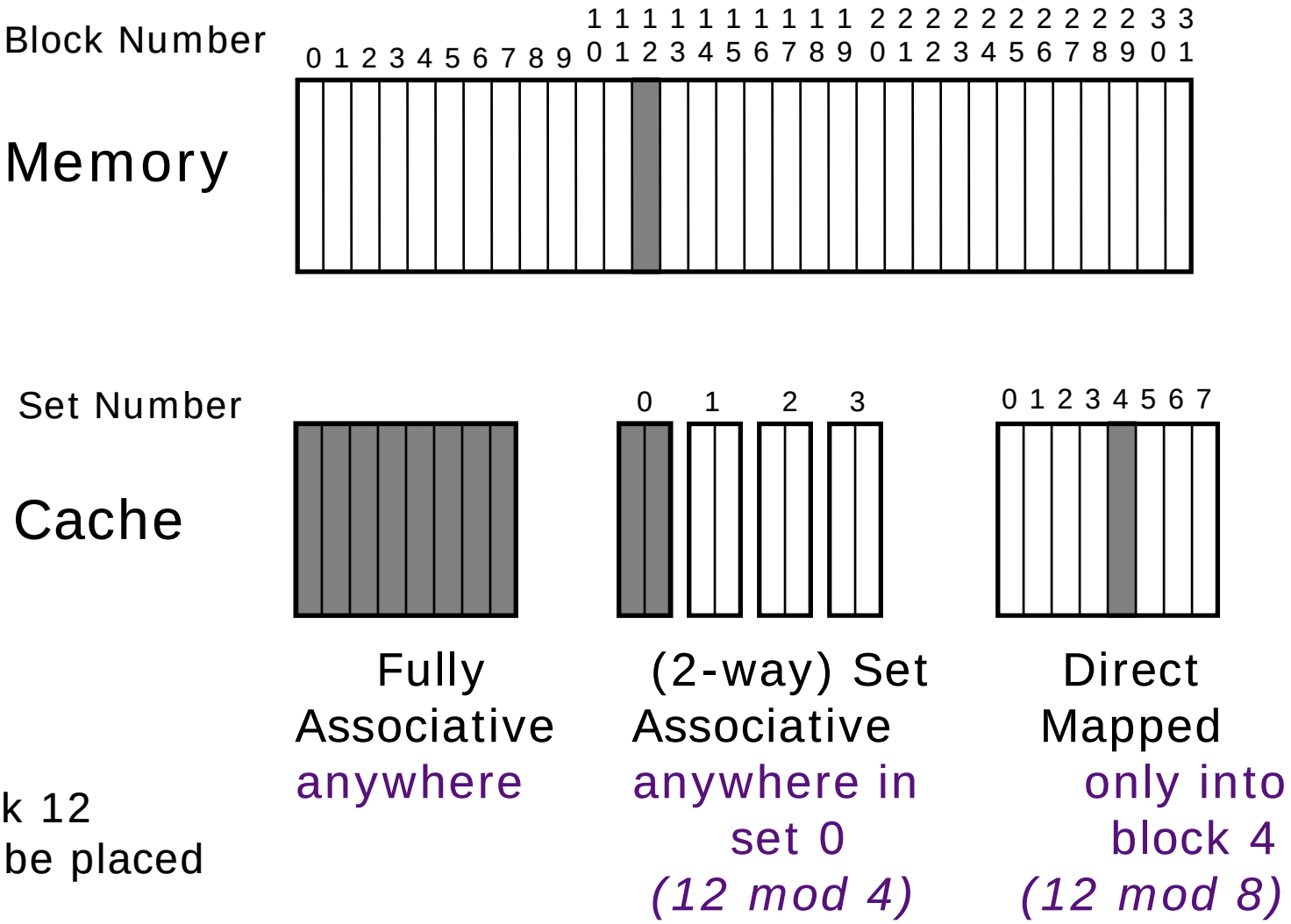
Read block of data from Main Memory

Wait ...

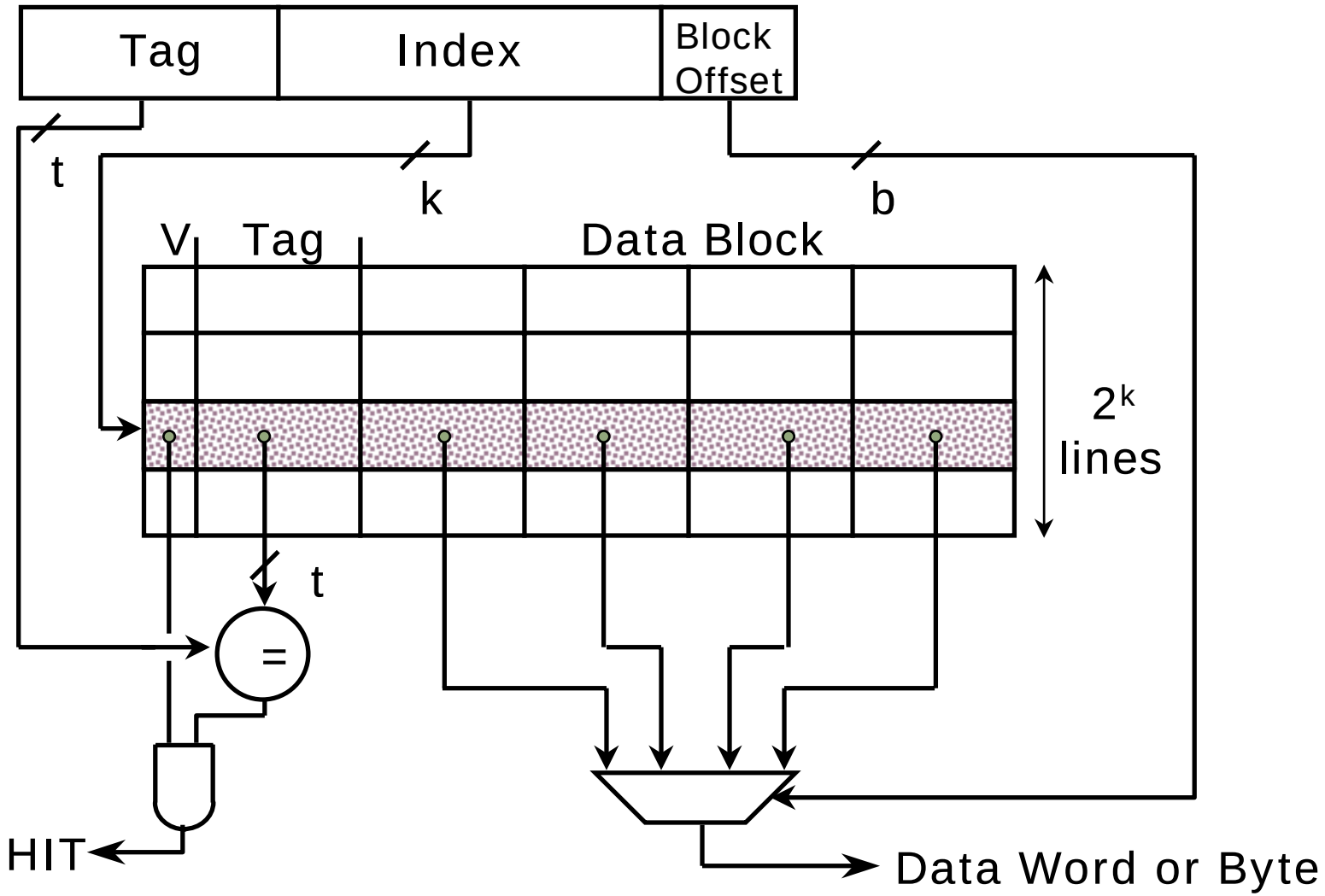
Return data to processor and update cache

Q: Which line do we replace?

Placement Policy

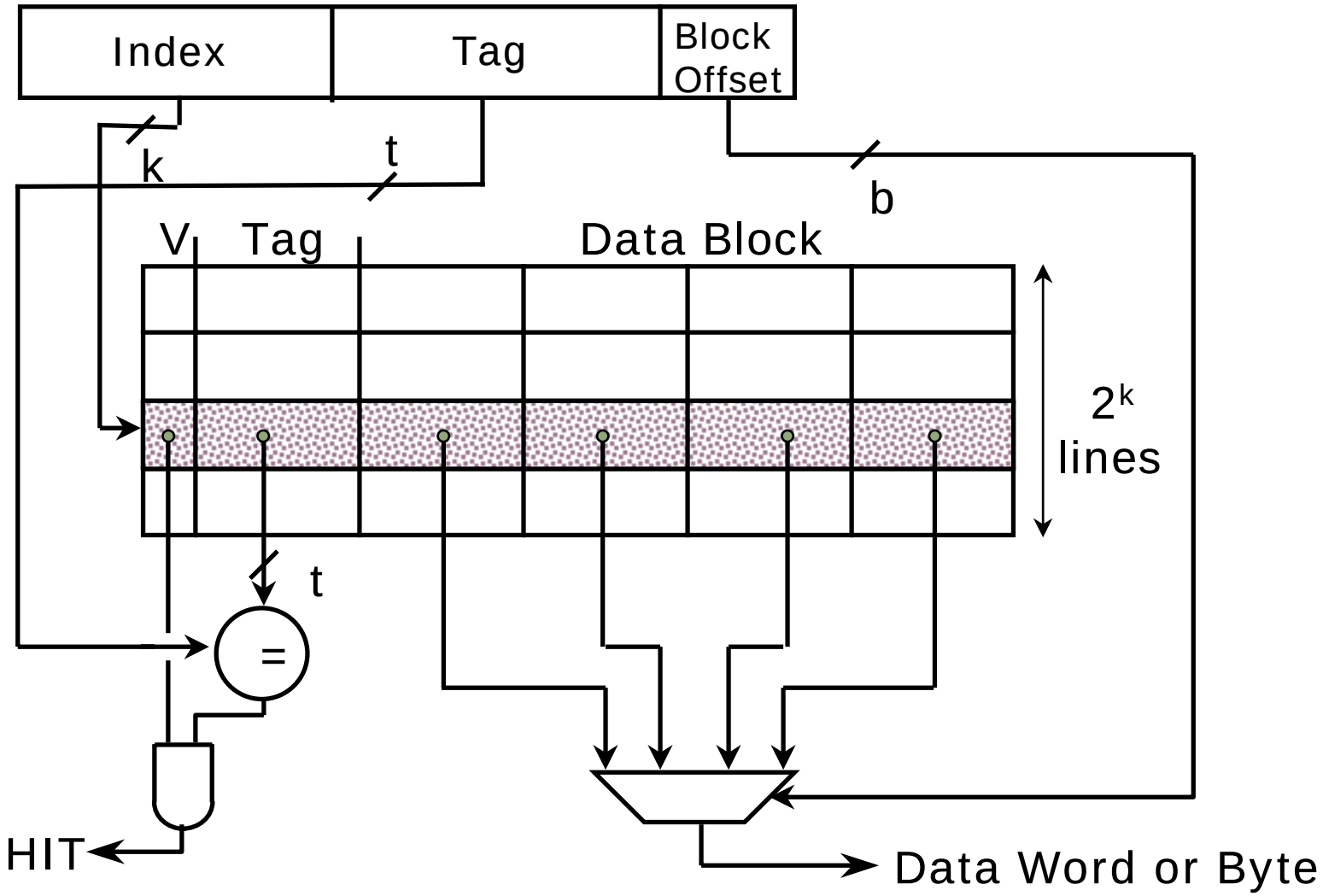


Direct-Mapped Cache

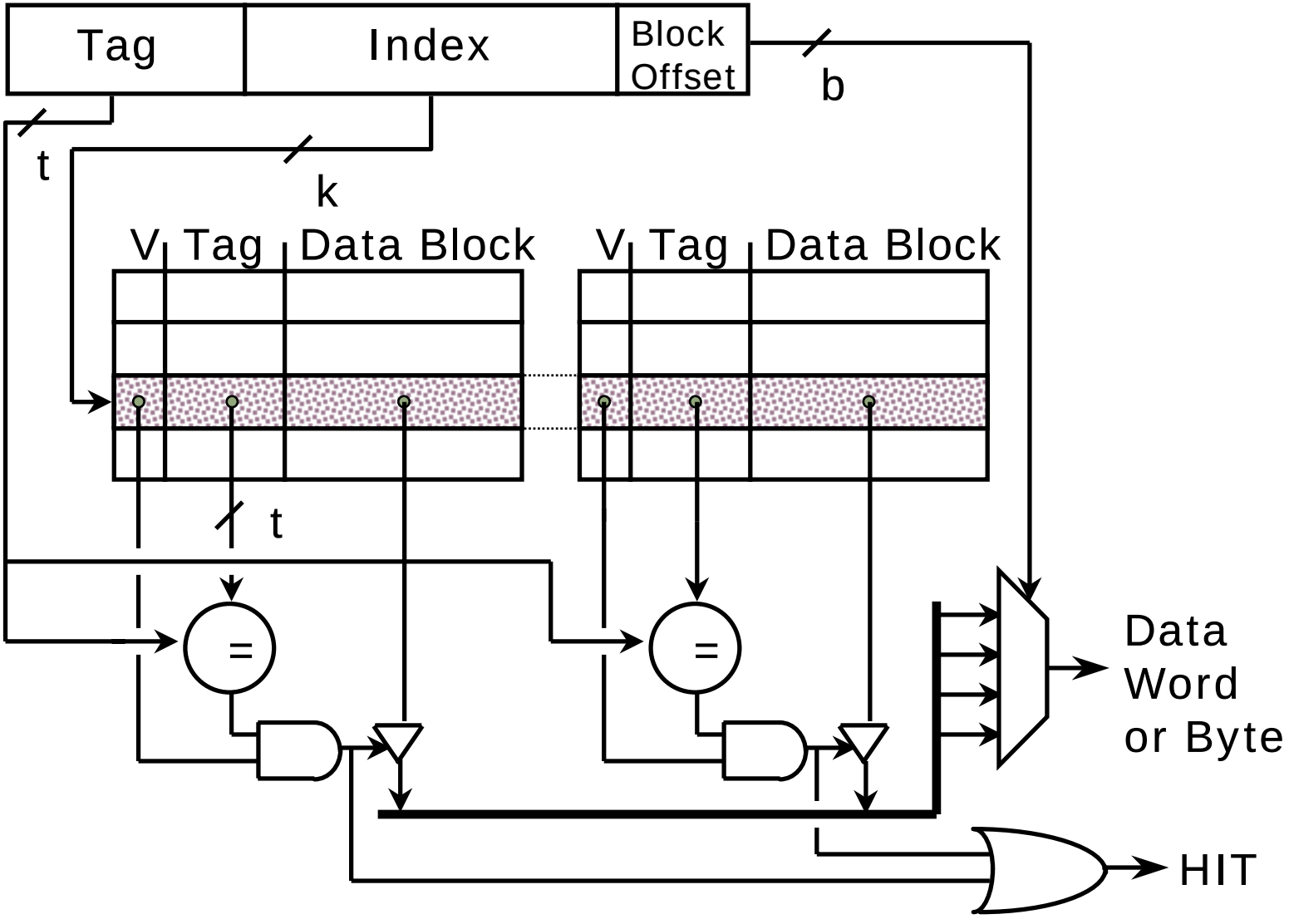


Direct Map Address Selection

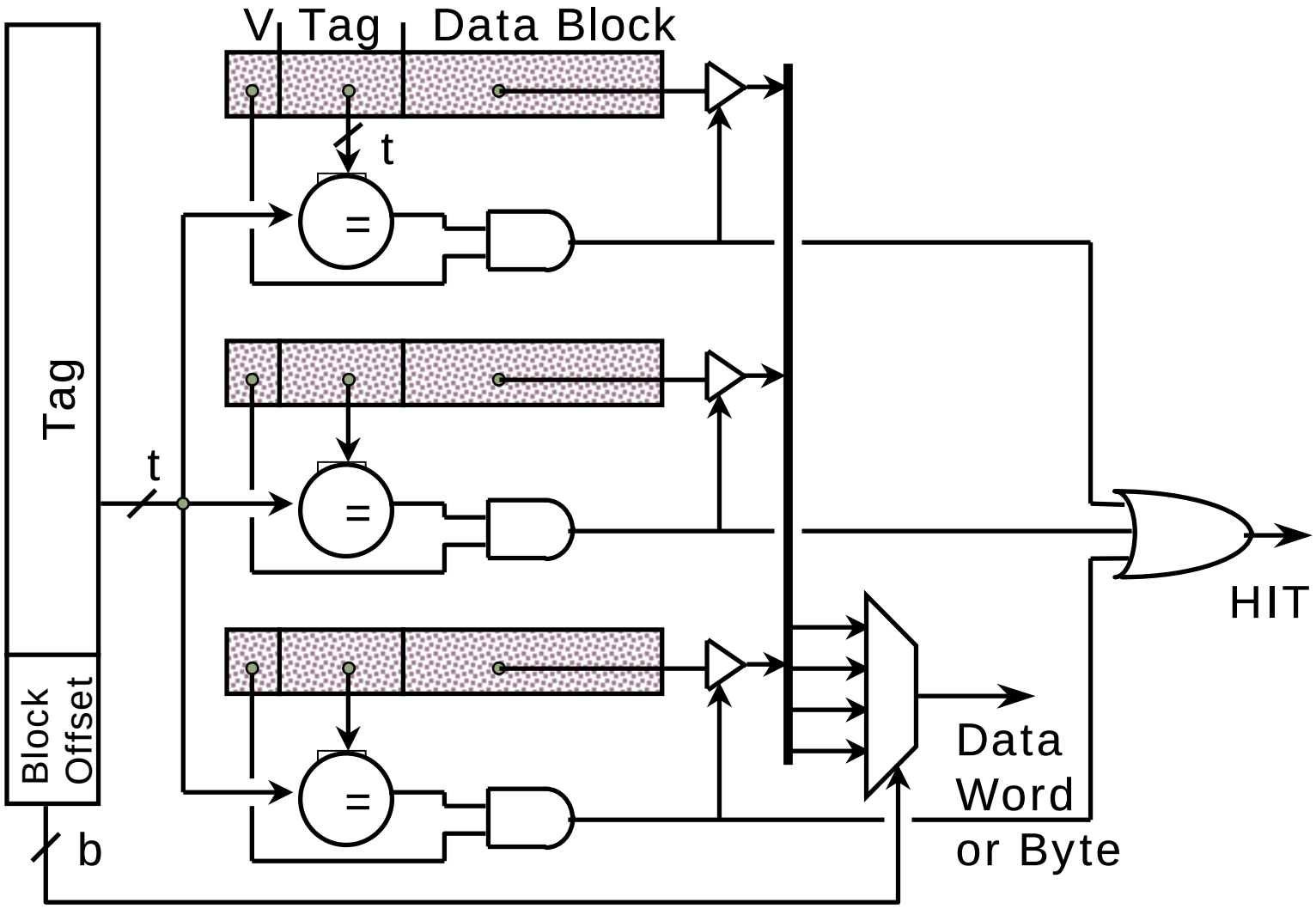
higher-order vs. lower-order address bits



2-Way Set-Associative Cache



Fully Associative Cache



Improving Cache Performance

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

To improve performance:

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

What is the simplest design strategy?

Biggest cache that doesn't increase hit time past 1-2 cycles (approx 8-32KB in modern technology)

[design issues more complex with out-of-order superscalar processors]

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 the program needs
  - misses that would occur even under perfect placement & replacement policy

- *Conflict:*
  misses from collisions due to block-placement strategy
  - misses that would not occur with full associativity

Effect of Cache Parameters on Performance

- Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

- Higher associativity
  + reduces conflict misses
  - may increase hit time

- Larger block size
  + reduces compulsory and capacity (reload) misses
  + exploit burst transfers in memory and on buses
  - increases conflict misses and miss penalty

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 (aka sector cache)
  - 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***		***0***		***0***	
0		***1***		***0***		***1***	

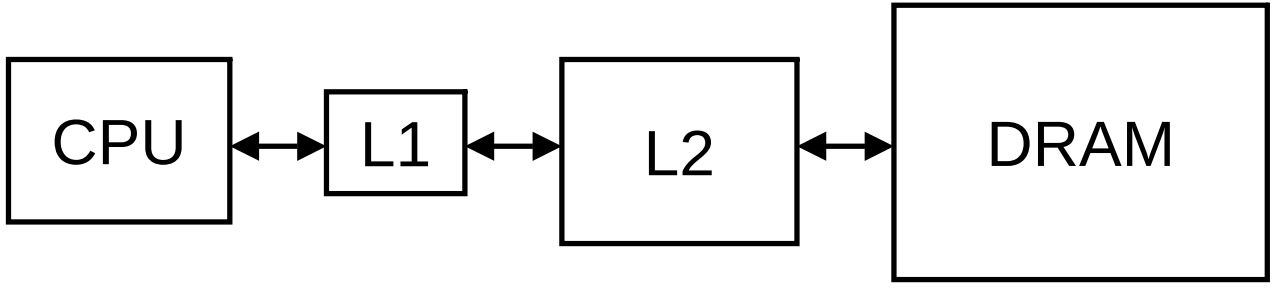
Replacement Policy

Which block from a set should be evicted?

- Random
- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - 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
- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block or blocks
- One-bit LRU
  - Each way represented by a bit. Set on use, replace first unused.

Multilevel Caches

- A memory cannot be large and fast
- Increasing sizes of cache at each level



Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions

Inclusion Policy

- Inclusive multilevel cache:
  - Inner cache holds copies of data in outer cache
  - External access need only check outer cache
  - Most common case
- *Exclusive* multilevel caches:
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

Why choose one type or the other?

Write Policies (2 separate questions)

Ql: propagate new value to memory?

- yes: write-through (propagate all writes to memory)
  - – update traffic independent of cache performance (bad)
  - – update/reference = f_{write}
- no: write-back (update memory on block replacement)
  - • set dirty bit on write, replace “clean” blocks w/o update
  - + less traffic for larger caches (low miss rates)
  - + multiple writes to same line combined into 1 update
  - + update/reference = $f_{dirty} * \%_{miss} *$ block size

Q2: on a miss, allocate a cache block?

- yes: write-allocate (usually with write-back)
- no: no-write-allocate (usually with write-through)

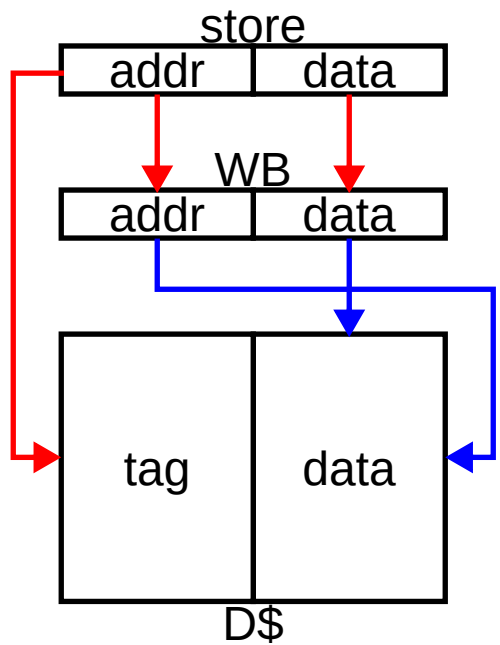
Write Buffers

parallel data access/tag check only for reads!!

- cannot be done for writes

solution: write buffer (WB)

- small (<4 entry) buffer
- allows writes to be *pipelined*
  - check tag
  - write store data into WB
  - write data from WB to cache (tags OK now)
  - in parallel!!
- problem: read to data in write buffer
  - match read addresses to WB addresses
  - stall or bypass



Write Buffers

can keep writes in write buffer for a while

- reads can still proceed (as described)
- coalesce (combine) writes to same block
  + fewer writes, fast burst transfer (transfer of entire block)

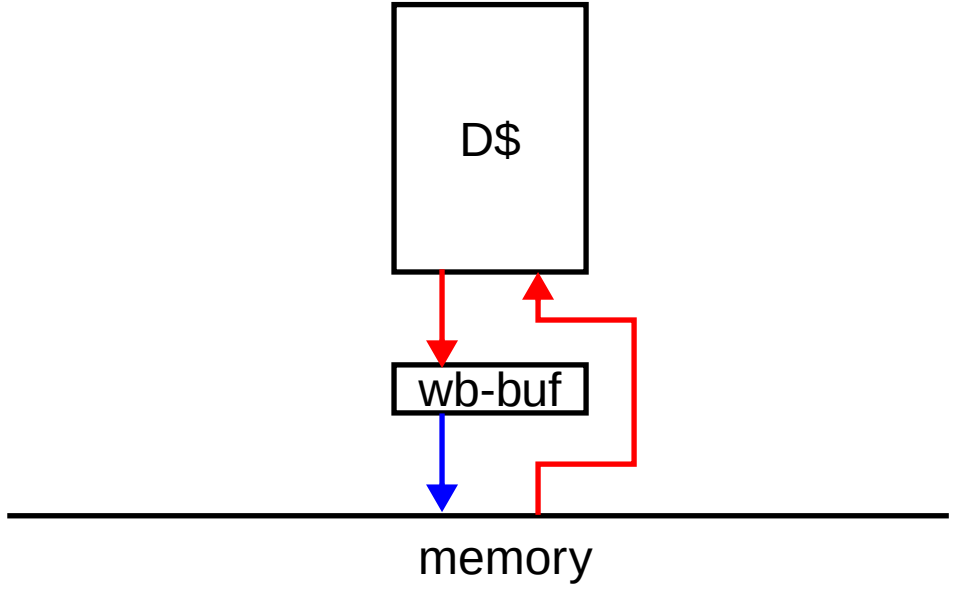
+ also good for write-thru caches (don't stall for memory)

write policy	hit/miss	writes to
back/alloc	both	cache
back/no-alloc	hit	cache
back/no-alloc	miss	memory
thru/alloc	both	both
thru/no-alloc	hit	both
thru/no-alloc	miss	memory

Writeback Buffers

like a write buffer, but between write-back cache and memory

+ allows replacement to be performed before writeback
  - replacement is more latency critical (someone is waiting)



Advanced Caches

- evaluation methods
- reducing miss rate
- reducing miss penalty
- reducing hit time

Evaluation Methods

- hardware counters
  - + accurate, realistic workloads (system, user, everything)
  - – machine must exist, can't vary cache parameters, non-deterministic
- analytical models (mathematical expressions)
  - + fast, good insight (can vary parameters)
  - – suspect accuracy, finding parameters is difficult
- trace-driven simulation (get $\%_{miss}$, plug t_{hit}, t_{miss})
  - + experiments repeatable, can be accurate
  - – time consuming, don't model speculative execution
- full processor simulation
  - + true performance (parallel misses, prefetches, speculation effects)
  - – complicated simulation model, very time consuming

Software Prefetching

- binding: prefetch into register (e.g., software pipelining)
  - + no ISA support, use normal loads
  - – need more registers, what about faults?
- non-binding: prefetch into cache (e.g., below)
  - – need ISA support (non-binding, non-faulting loads)
  - + simpler semantics, preferred

```
for (j=0; j<COLS; j++)
   for (i=0; i<ROWS; i+=BLOCK_SIZE)
     prefetch (&x[i][j]+BLOCK_SIZE);
     for (ii=i; ii<i+BLOCK_SIZE-1; ii++)
        sum += x[ii][j];
```

Hardware Prefetching

what to prefetch?

- one block ahead (spatially)
  + works well for instructions, sequential data (arrays)
- more complex: use “address prediction”
  - needed for non-sequential data

when to prefetch?

- on every reference
- on a miss (i.e., effectively double block size)
  + better performance than doubling block size (why?)
- when resident block becomes dead [ISCA’01]
  - when no one will use it anymore (how do we know this?)