

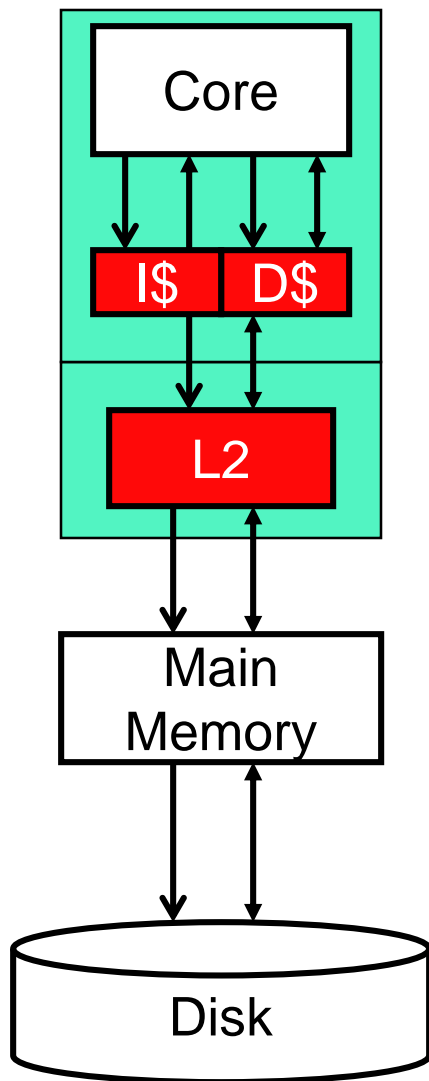
Modern Computer Architecture

with sources that included UPenn & University of Wisconsin slides
by Joe Devietti, Milo Martin & Amir Roth, Mark Hill, Guri Sohi, Jim Smith, and David Wood

Schedule

- Introduction and Transistors
- Parallel computing (Isaac D. Scherson, University of California, Irvine, in October)
- ISAs
- Performance
- Pipelining Basic
- Branch Prediction
- Caches
- Virtual Memory
- Out-of-Order Execution
- Multicore multi-thread
- Vectors/GPUs for data parallelism.

Caches



- “Cache”: hardware managed
 - Hardware automatically retrieves missing data
 - Built from fast on-chip SRAM
 - In contrast to off-chip, DRAM “main memory”
- **Average access time** of a memory component
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure
→ memory hierarchy
- Cache ABCs (**associativity, block size, capacity**)
- **Performance optimizations**
 - Prefetching & data restructuring
- **Handling writes**
 - Write-back vs. write-through
- **Memory hierarchy**
 - Smaller, faster, expensive → bigger, slower, cheaper

Motivation

- Processor can compute only as fast as memory
 - A 3Ghz processor can execute an “add” operation in 0.33ns
 - Today’s “main memory” latency is more than 33ns
 - Naïve implementation:
 - loads/stores can be 100x slower than other operations
- Unobtainable goal:
 - Memory that operates at processor speeds
 - Memory as large as needed for all running programs
 - Memory that is cost effective
- Can’t achieve all of these goals at once
 - Example: latency of an SRAM is at least: $\sqrt{\text{number of bits}}$

Memories (SRAM & DRAM)

Types of Memory

- **Static RAM (SRAM)**

- 6 or 8 transistors per bit
 - Two inverters (4 transistors) + transistors for reading/writing
- Optimized for speed (first) and density (second)
- Fast (sub-nanosecond latencies for small SRAM)
 - Speed roughly proportional to its area ($\sim \sqrt{\text{number of bits}}$)
- Mixes well with standard processor logic

- **Dynamic RAM (DRAM)**

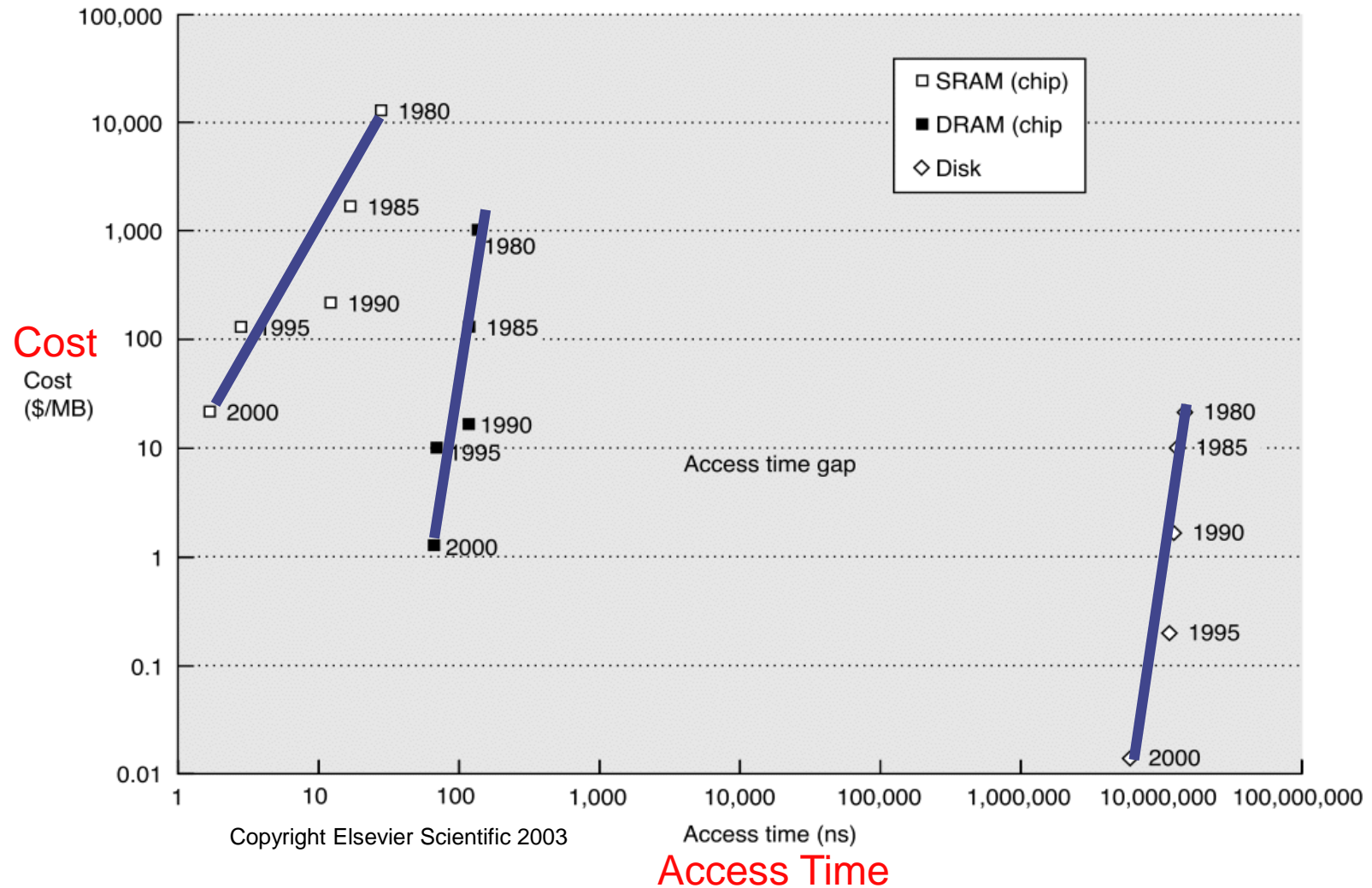
- 1 transistor + 1 capacitor per bit
- Optimized for density (in terms of cost per bit)
- Slow ($>30\text{ns}$ internal access, $\sim 50\text{ns}$ pin-to-pin)
- Different fabrication steps (does not mix well with logic)

- Nonvolatile storage: Magnetic disk, Flash RAM

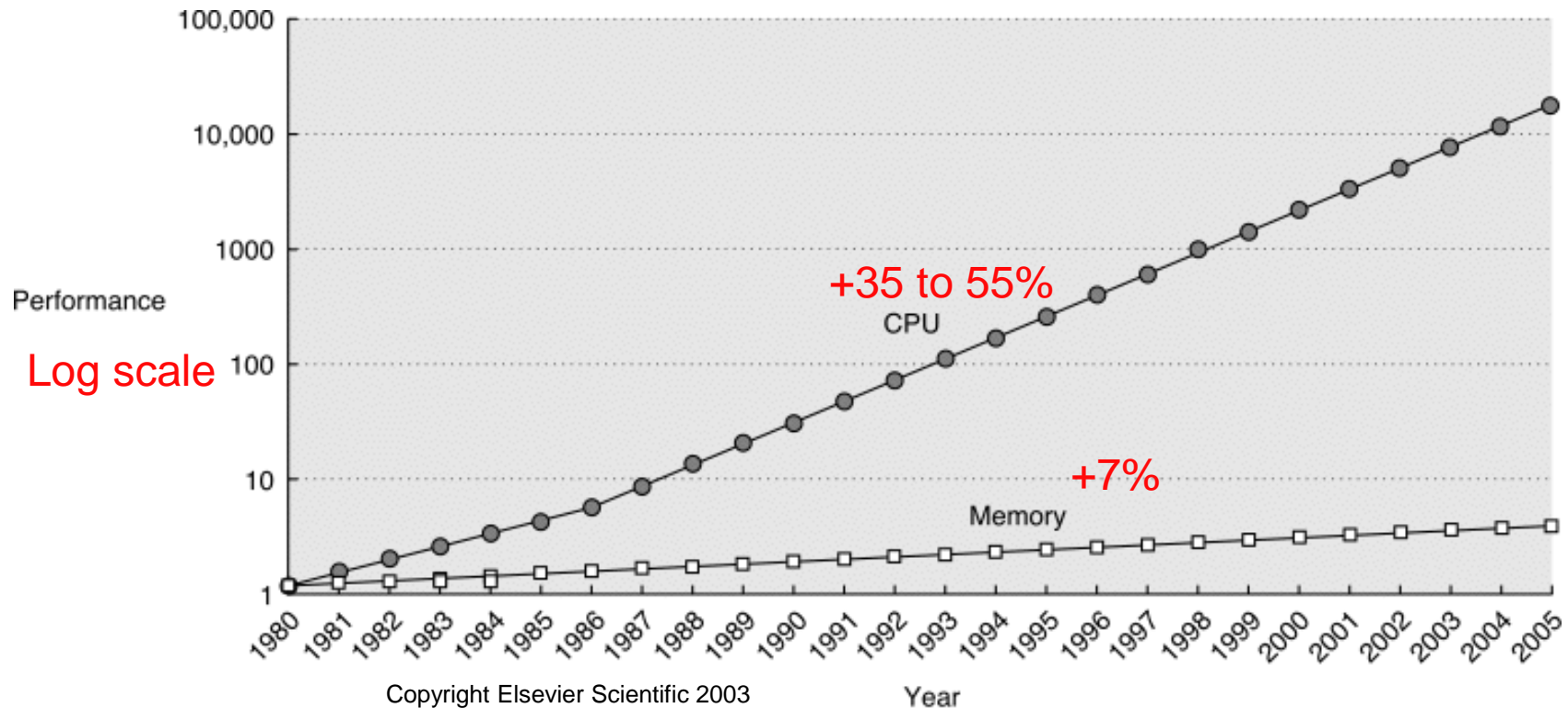
Memory & Storage Technologies

- **Cost** - what can \$200 buy (2009)?
 - SRAM: 16MB
 - DRAM: 4,000MB (4GB) – 250x cheaper than SRAM
 - Flash: 64,000MB (64GB) – 16x cheaper than DRAM
 - Disk: 2,000,000MB (2TB) – 32x vs. Flash (512x vs. DRAM)
- **Latency**
 - SRAM: <1 to 2ns (on chip)
 - DRAM: ~50ns – 100x or more slower than SRAM
 - Flash: 75,000ns (75 microseconds) – 1500x vs. DRAM
 - Disk: 10,000,000ns (10ms) – 133x vs Flash (200,000x vs DRAM)
- **Bandwidth**
 - SRAM: 300GB/sec (e.g., 12-port 8-byte register file @ 3Ghz)
 - DRAM: ~25GB/s
 - Flash: 0.25GB/s (250MB/s), 100x less than DRAM
 - Disk: 0.1 GB/s (100MB/s), 250x vs DRAM, **sequential** access only

Memory Technology Trends



The “Memory Wall”



- Processors get faster more quickly than memory (note log scale)
 - Processor speed improvement: 35% to 55%
 - Memory latency improvement: 7%

The Memory Hierarchy

Known From the Beginning

“Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible.”

Burks, Goldstine, VonNeumann

“Preliminary discussion of the logical design of an electronic computing instrument”

IAS memo 1946

Big Observation: Locality & Caching

- **Locality of memory references**
 - Empirical property of real-world programs, few exceptions
- **Temporal locality**
 - Recently referenced data is likely to be referenced again soon
 - **Reactive**: “cache” recently used data in small, fast memory
- **Spatial locality**
 - More likely to reference data near recently referenced data
 - **Proactive**: “cache” large chunks of data to include nearby data
- Both properties hold for both data and instructions
- Cache: “Hash table” of recently used blocks of data
 - In hardware, finite-sized, transparent to software

Spatial and Temporal Locality Example

- Which memory accesses demonstrate spatial locality?
- Which memory accesses demonstrate temporal locality?

```
int sum = 0;
int x[1000];

for(int c = 0; c < 1000; c++) {
    sum += c;

    x[c] = 0;
}
```

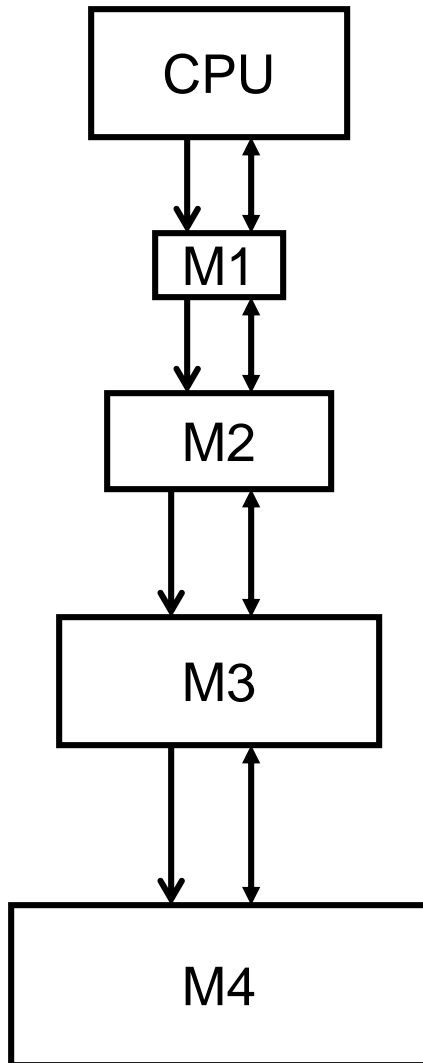
Library Analogy

- Consider books in a library
- Library has lots of books, but it is slow to access
 - Far away (time to walk to the library)
 - Big (time to walk within the library)
- How can you avoid these latencies?
 - Check out books, take them home with you
 - Put them on desk, on bookshelf, etc.
 - But desks & bookshelves have limited capacity
 - Keep recently used books around (**temporal locality**)
 - Grab books on related topic at the same time (**spatial locality**)
 - Guess what books you'll need in the future (prefetching)

Library Analogy Explained

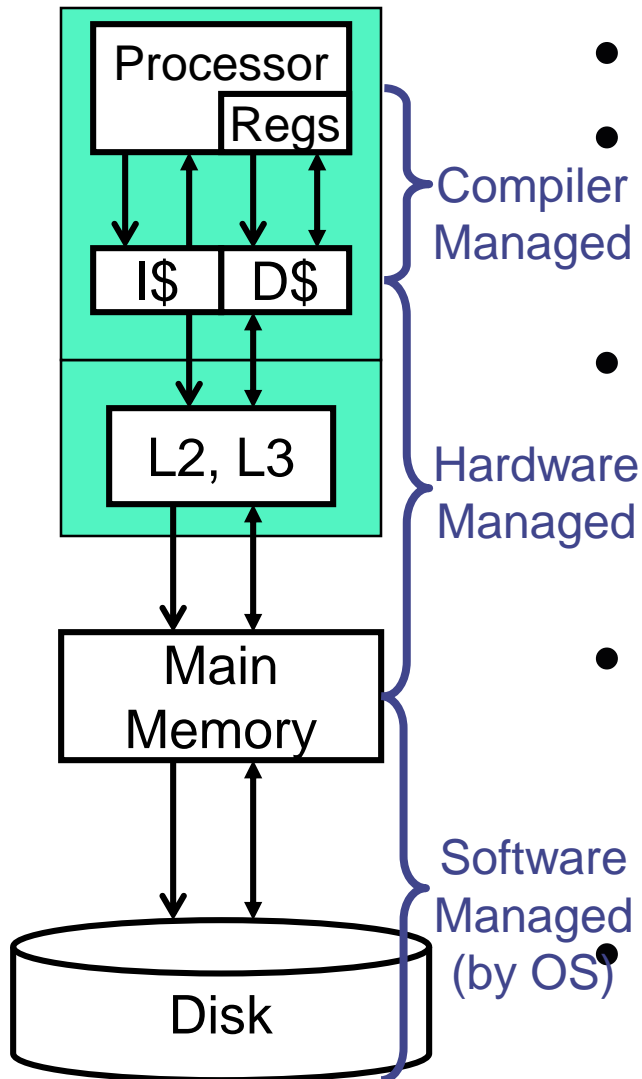
- Registers \leftrightarrow books on your desk
 - Actively being used, small capacity
- Caches \leftrightarrow bookshelves
 - Moderate capacity, pretty fast to access
- Main memory \leftrightarrow library
 - Big, holds almost all data, but slow
- Disk (virtual memory) \leftrightarrow inter-library loan
 - Very slow, but hopefully really uncommon

Exploiting Locality: Memory Hierarchy



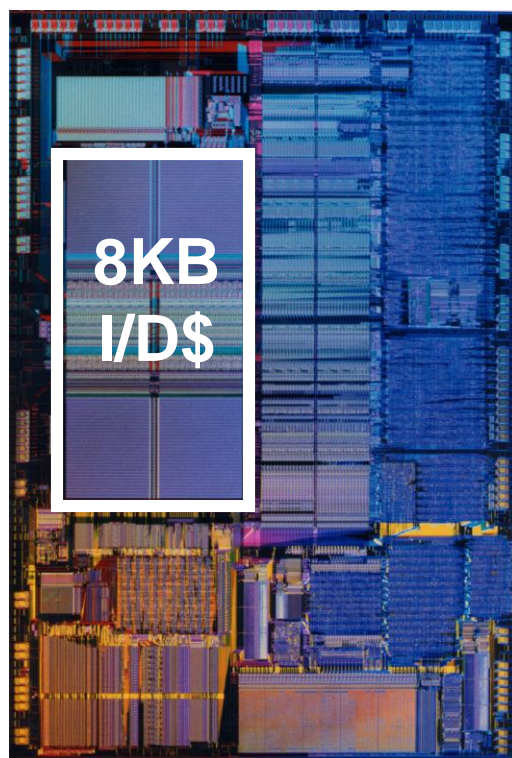
- Hierarchy of memory components
 - Upper components
 - Fast ↔ Small ↔ Expensive
 - Lower components
 - Slow ↔ Big ↔ Cheap
- Connected by “buses”
 - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
 - M1 + next most frequently accessed in M2, etc.
 - Move data up-down hierarchy
- Optimize average access time
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Attack each component

Concrete Memory Hierarchy

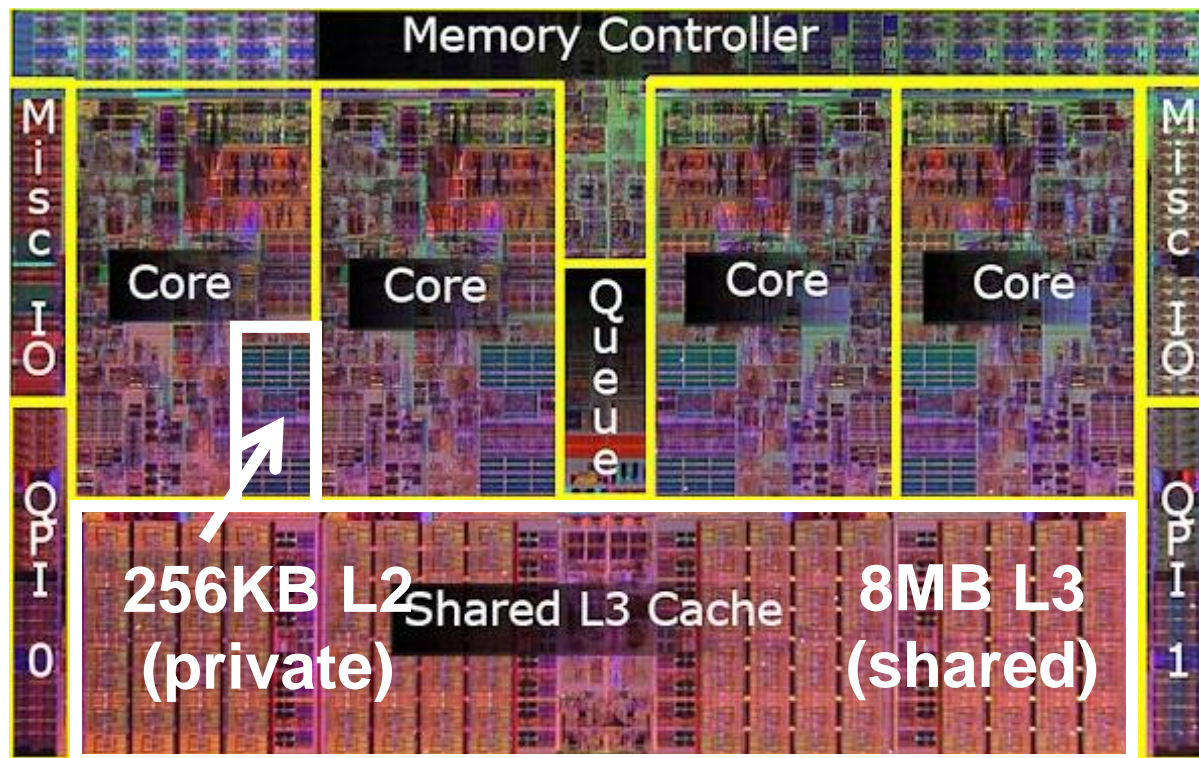


- 0th level: **Registers**
- 1st level: **Primary caches**
 - Split instruction (I\$) and data (D\$)
 - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
 - On-chip, typically made of SRAM
 - 2nd level typically ~256KB to 512KB
 - “Last level cache” typically 4MB to 16MB
- 3rd level: **main memory**
 - Made of DRAM (“Dynamic” RAM)
 - Typically 2-16GB for desktops/laptops
 - Servers can have >1 TB
- 4th level: **disk (swap and files)**
 - Uses magnetic disks or flash drives

Evolution of Cache Hierarchies



Intel 486



Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

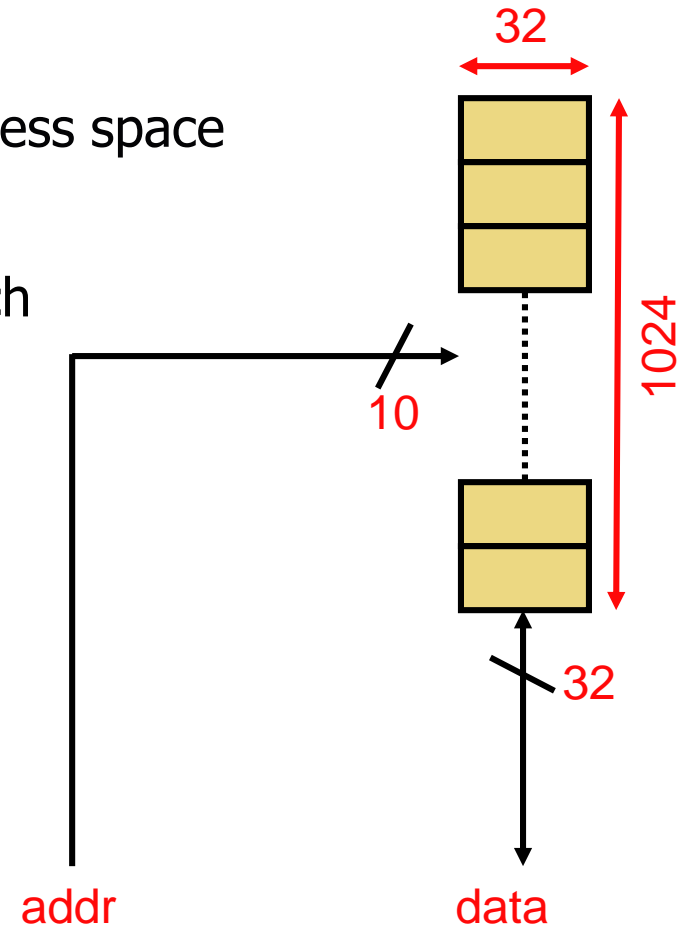
Caches

Analogy to a Software Hashtable

- What is a “hash table”?
 - What is it used for?
 - How does it work?
- Short answer:
 - Maps a “key” to a “value”
 - Constant time lookup/insert
 - Have a table of some size, say N , of “buckets”
 - Take a “key” value, apply a hash function to it
 - Insert and lookup a “key” at “hash(key) modulo N ”
 - Need to store the “key” and “value” in each bucket
 - Need to check to make sure the “key” matches
 - Need to handle conflicts/overflows somehow (chaining, re-hashing)

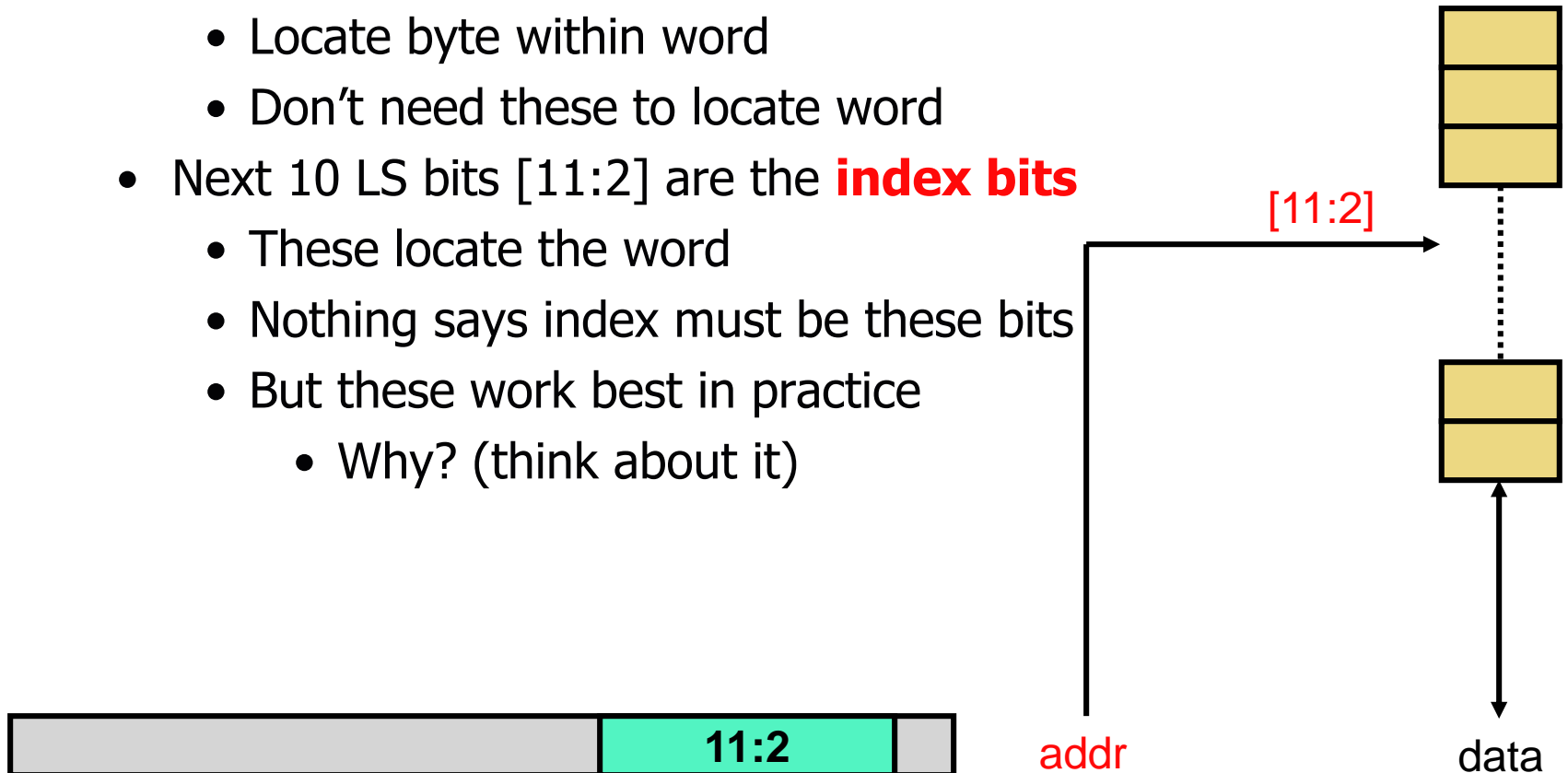
Hardware Cache Organization

- **Cache is a hardware hashtable**
 - with one exception: data not always present
- The setup
 - 32-bit ISA → 4B addresses, 2^{32} B address space
- Logical cache organization
 - 4KB, organized as 1K **blocks** of 4B each
 - Each block can hold a 4B word
- Physical cache implementation
 - 1K (1024 bit) by 4B **SRAM**
 - Called **data array**
 - 10-bit address input
 - 32-bit data input/output



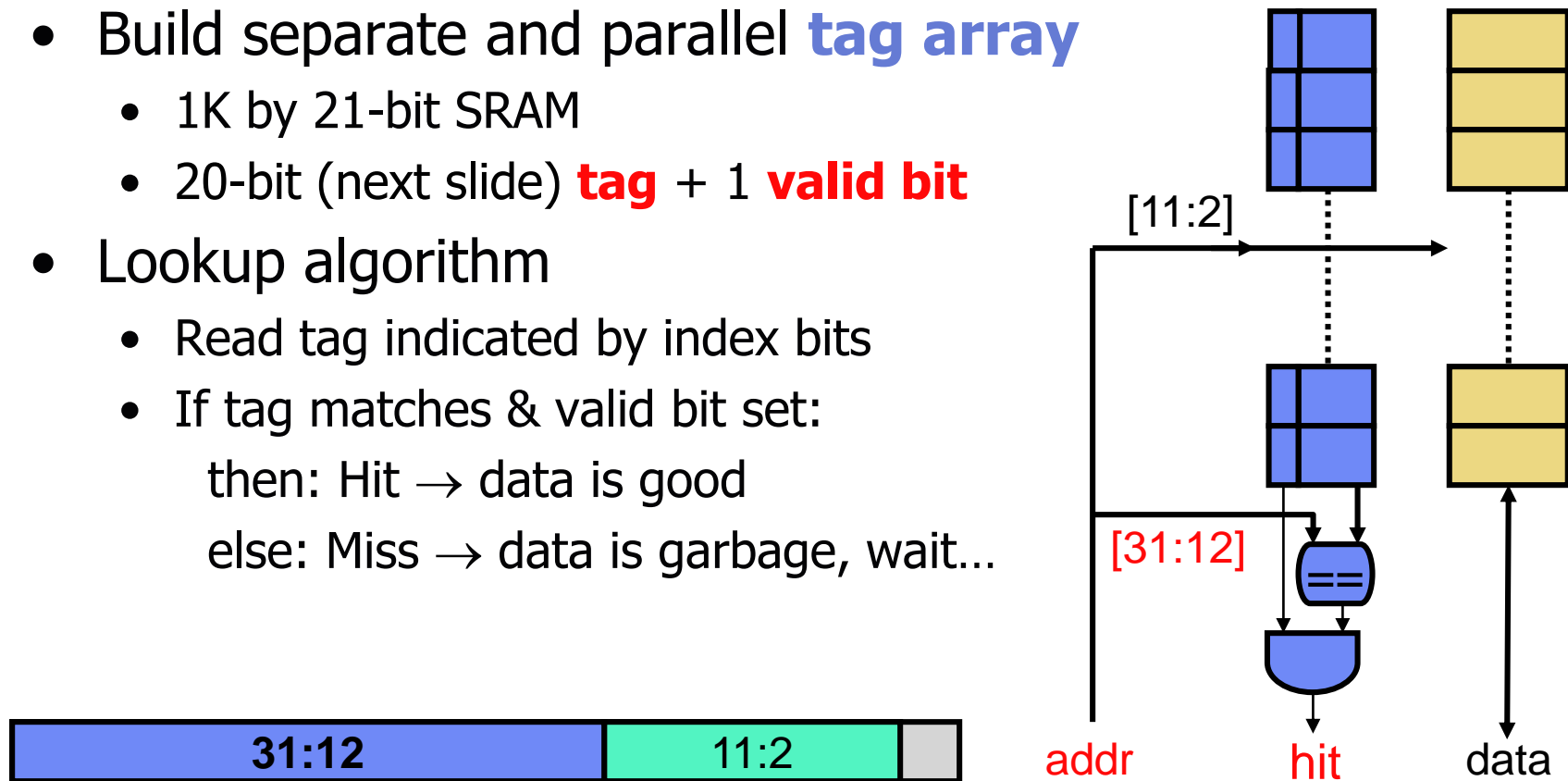
Looking Up A Block

- Q: which 10 of the 32 address bits to use?
- A: bits [11:2]
 - 2 least significant (LS) bits [1:0] are the **offset bits**
 - Locate byte within word
 - Don't need these to locate word
 - Next 10 LS bits [11:2] are the **index bits**
 - These locate the word
 - Nothing says index must be these bits
 - But these work best in practice
 - Why? (think about it)



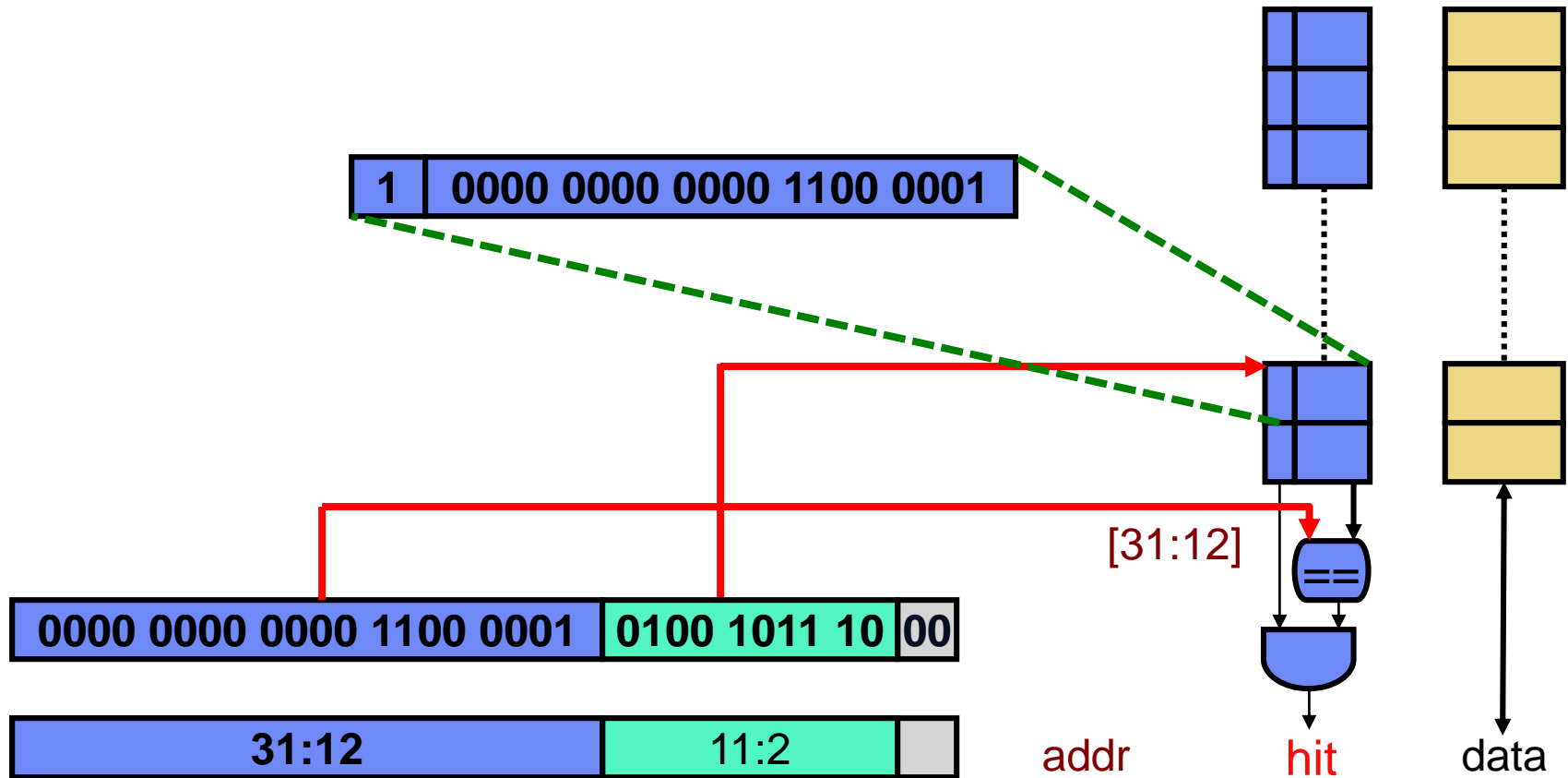
Knowing that You Found It

- 2^{20} blocks map to each cache row (**set**)
 - How to know which if any is currently there?
 - Tag each cache word with remaining address bits [31:12]
- Build separate and parallel **tag array**
 - 1K by 21-bit SRAM
 - 20-bit (next slide) **tag** + 1 **valid bit**
- Lookup algorithm
 - Read tag indicated by index bits
 - If tag matches & valid bit set:
 - then: Hit → data is good
 - else: Miss → data is garbage, wait...



A Concrete Example

- Lookup address x000C14B8
 - Index = $\text{addr}[11:2] = (\text{addr} \gg 2) \& \text{x3FF} = \text{x12E}$
 - Tag = $\text{addr}[31:12] = (\text{addr} \gg 12) = \text{x000C1}$



Calculating Tag Overhead

- “32KB cache” means cache holds 32KB of data
 - Called **capacity**
 - Tag storage is considered overhead
- Tag overhead of 32KB cache with 1024 32B **frames**
 - frame is physical structure holding a block, frame size==block size
 - 32B frame → 5-bit offset
 - 1024 frames → 10-bit index
 - 32-bit address – 5-bit offset – 10-bit index = 17-bit tag
 - (17-bit tag + 1-bit valid) * 1024 frames = 18Kb tags = 2.2KB tags
 - ~6% overhead
- What about 64-bit addresses?
 - Tag increases to 49 bits, ~20% overhead (worst case)

Handling a Cache Miss

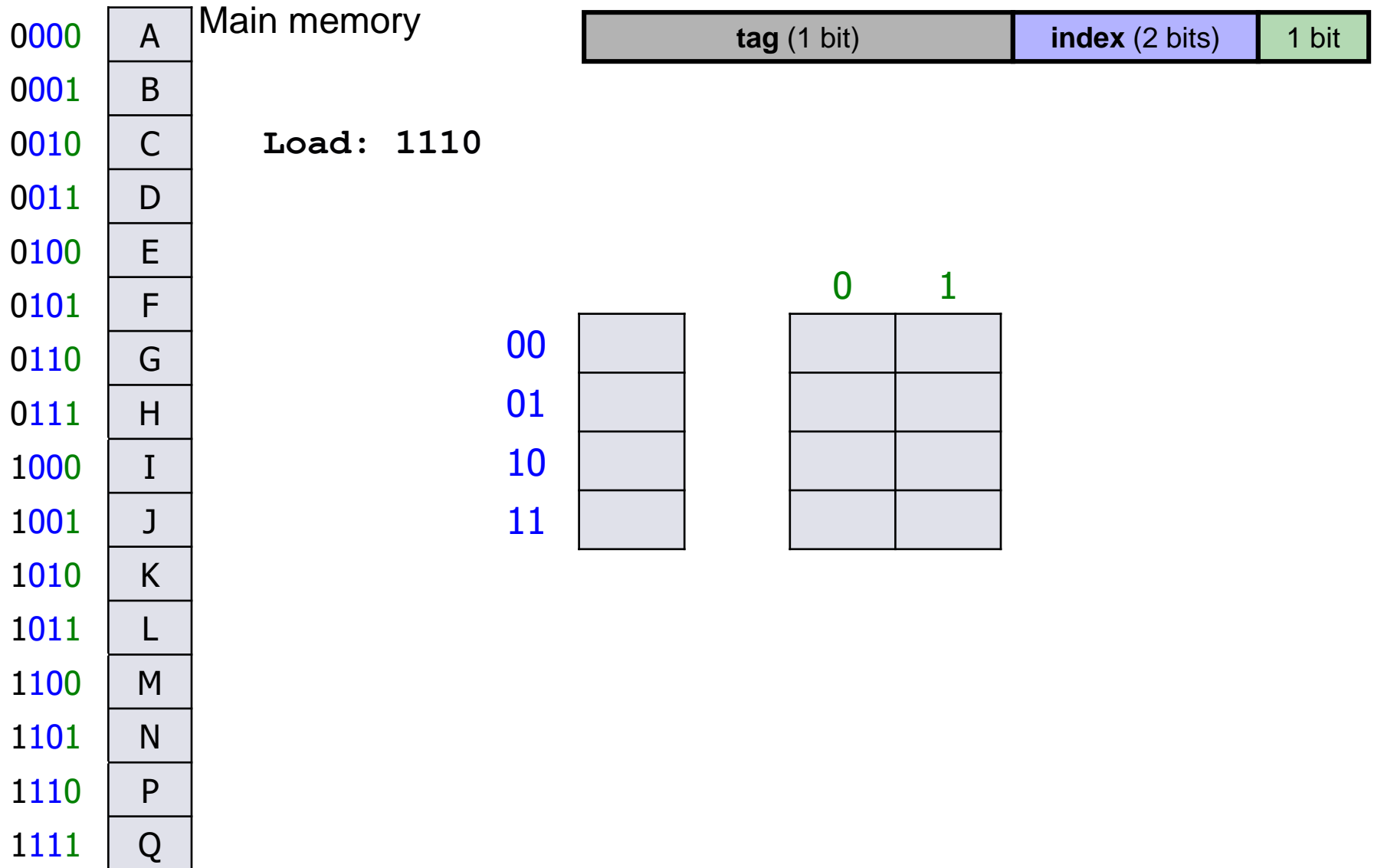
- What if requested data isn't in the cache?
 - How does it get in there?
- **Cache controller**: finite state machine
 - Remembers miss address
 - Accesses next level of memory
 - Waits for response
 - Writes data/tag into proper locations
- All of this happens on the **fill path**
- Sometimes called **backside**

Cache Examples

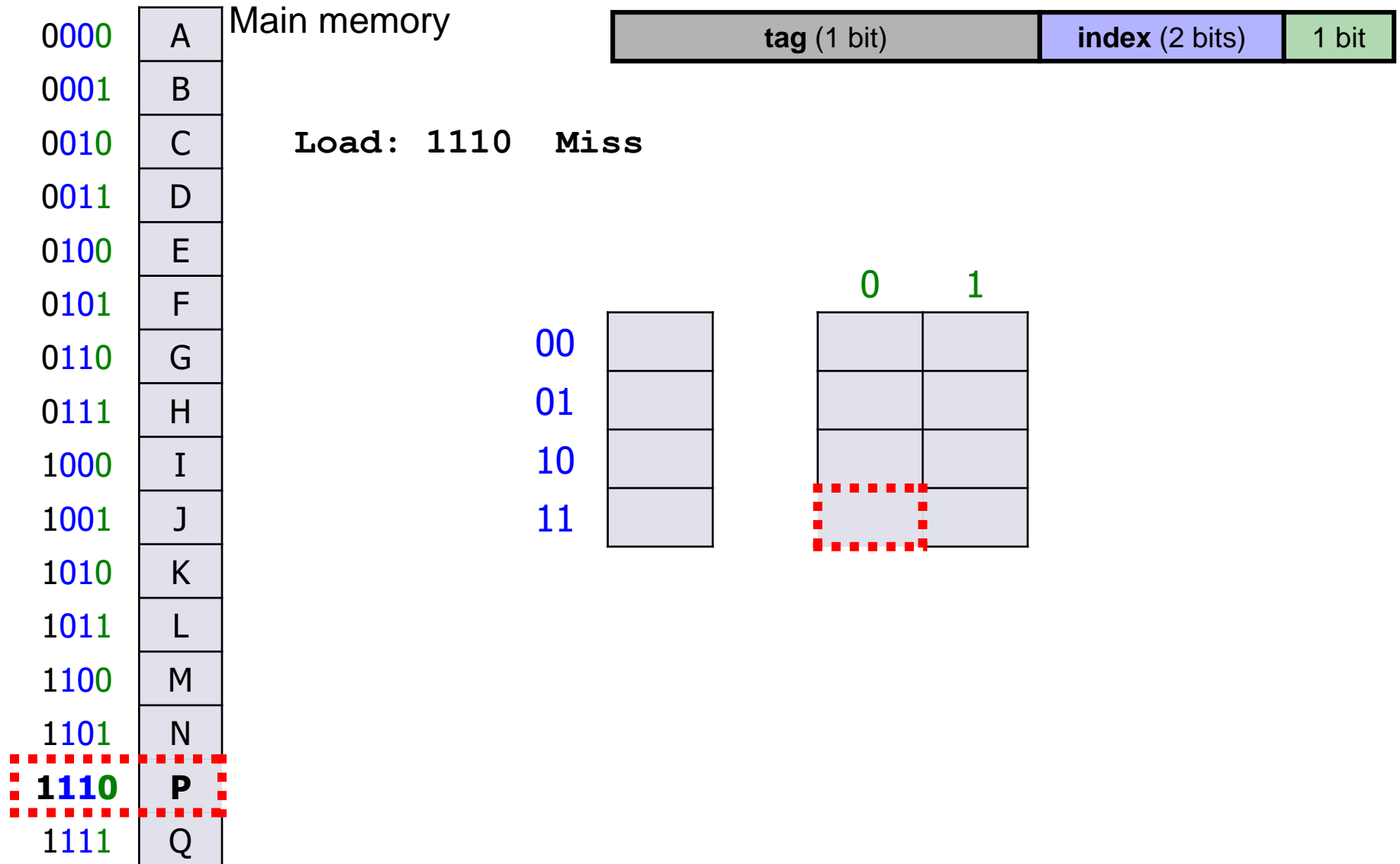
- 4-bit addresses \rightarrow 16B memory
 - Simpler cache diagrams than 32-bits
- 8B cache, 2B blocks
 - Figure out number of rows/sets
 - 4 (capacity / block-size)
 - Figure out how address splits into offset/index/tag bits
 - Offset: least-significant $\log_2(\text{block-size}) = \log_2(2) = 1 \rightarrow 000\mathbf{0}$
 - Index: next $\log_2(\text{number-of-sets}) = \log_2(4) = 2 \rightarrow 0\mathbf{00}0$
 - Tag: rest = $4 - 1 - 2 = 1 \rightarrow \mathbf{0}000$



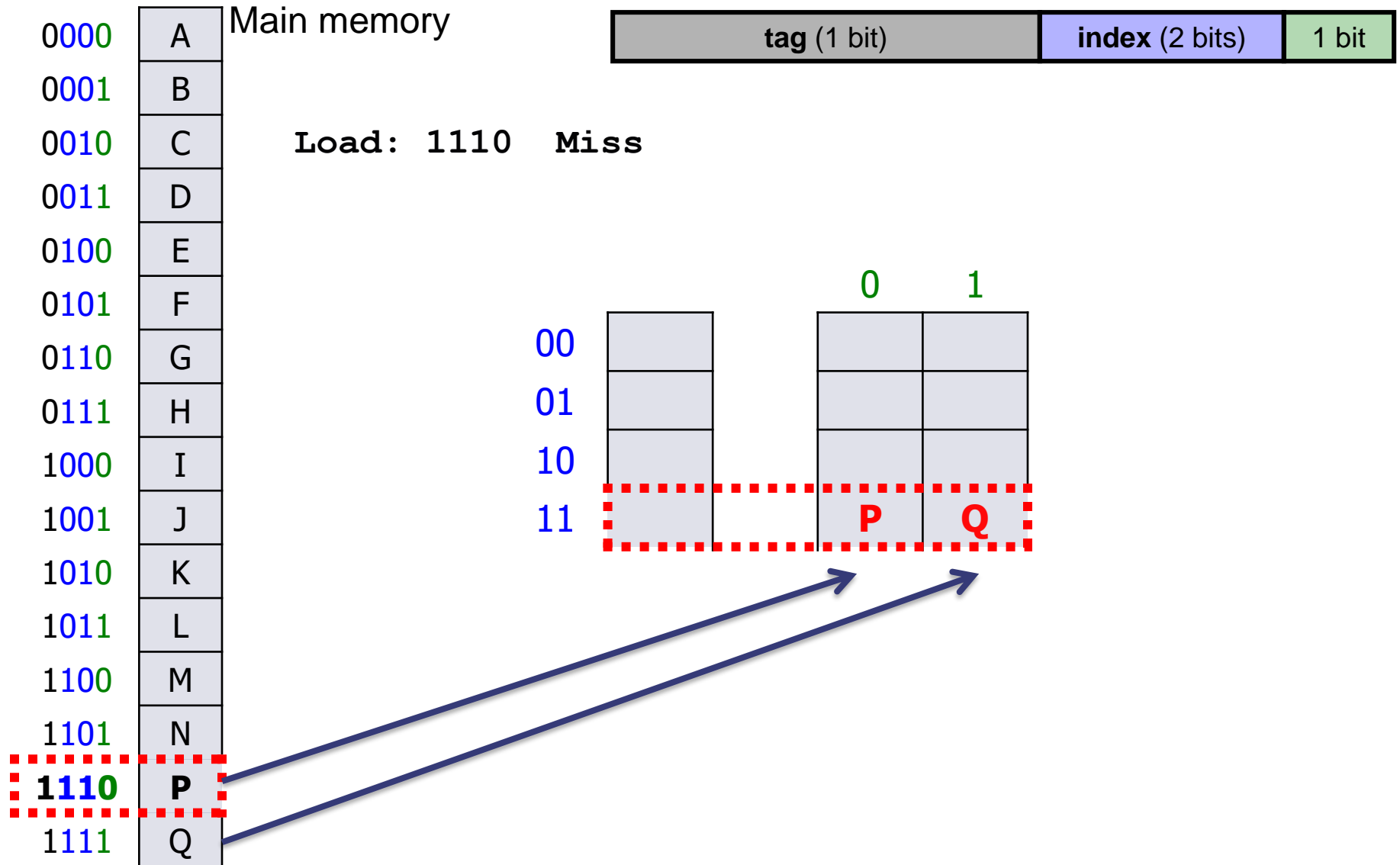
4-bit Address, 8B Cache, 2B Blocks



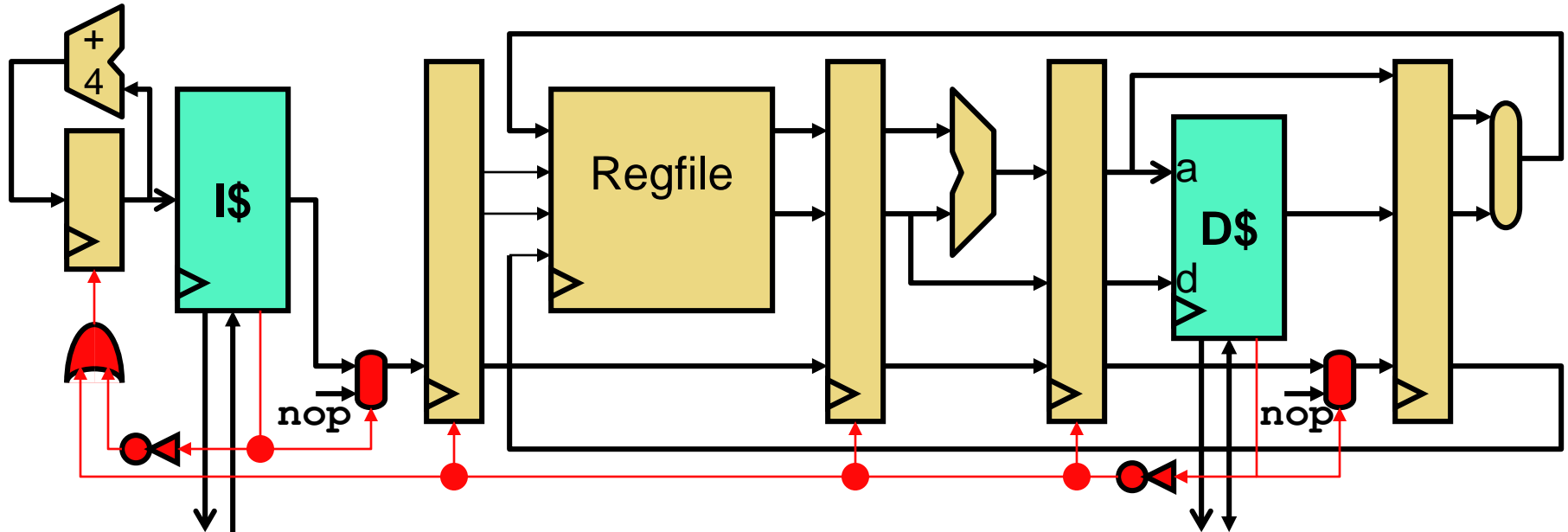
4-bit Address, 8B Cache, 2B Blocks



4-bit Address, 8B Cache, 2B Blocks

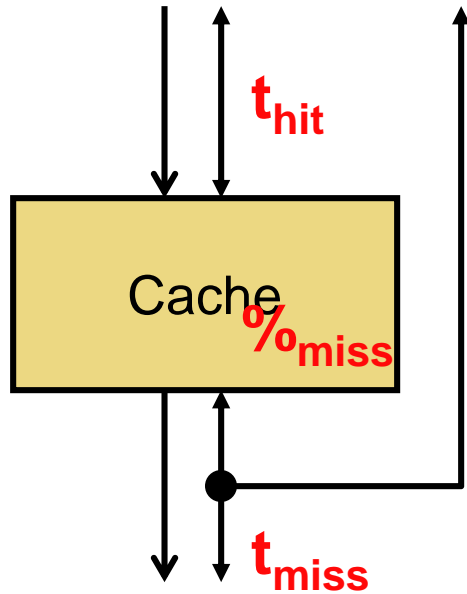


Cache Misses and Pipeline Stalls



- I\$ and D\$ misses stall pipeline just like data hazards
 - Stall logic driven by miss signal
 - Cache “logically” re-evaluates hit/miss every cycle
 - Block is filled → miss signal de-asserts → pipeline restarts

Cache Performance Equation



- For a cache
 - **Access**: read or write to cache
 - **Hit**: desired data found in cache
 - **Miss**: desired data not found in cache
 - Must get from another component
 - No notion of "miss" in register file
 - **Fill**: action of placing data into cache
 - $\%_{miss}$ (miss-rate): #misses / #accesses
 - t_{hit} : time to read data from (write data to) cache
 - t_{miss} : time to read data into cache
- Performance metric: average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

CPI Calculation with Cache Misses

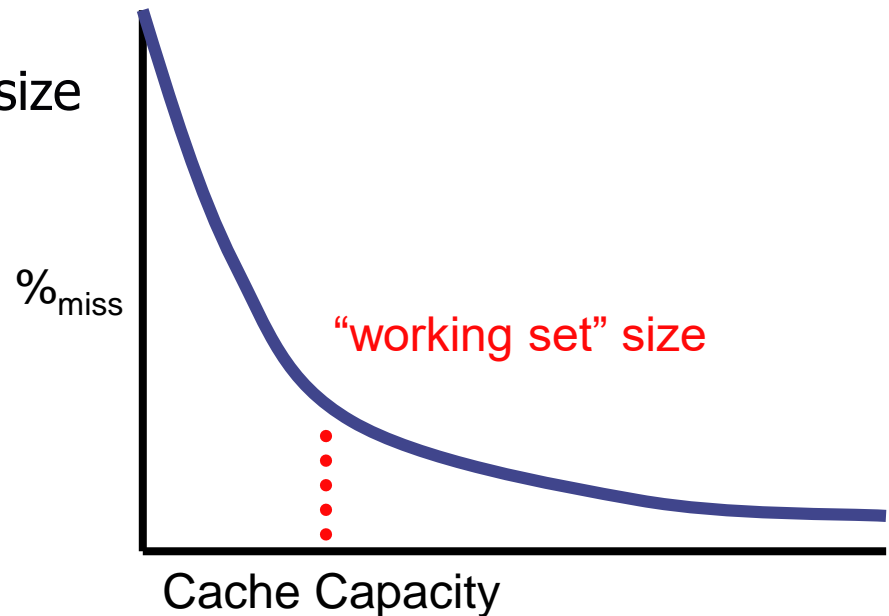
- Parameters
 - Simple pipeline with base CPI of 1
 - Instruction mix: 30% loads/stores
 - I\$: $\%_{\text{miss}} = 2\%$, $t_{\text{miss}} = 10$ cycles
 - D\$: $\%_{\text{miss}} = 10\%$, $t_{\text{miss}} = 10$ cycles
- What is new CPI?
 - $\text{CPI}_{\text{I\$}} = \%_{\text{missI\$}} * t_{\text{miss}} = 0.02 * 10 \text{ cycles} = 0.2 \text{ cycle}$
 - $\text{CPI}_{\text{D\$}} = \%_{\text{load/store}} * \%_{\text{missD\$}} * t_{\text{missD\$}} = 0.3 * 0.1 * 10 \text{ cycles} = 0.3 \text{ cycle}$
 - $\text{CPI}_{\text{new}} = \text{CPI} + \text{CPI}_{\text{I\$}} + \text{CPI}_{\text{D\$}} = 1 + 0.2 + 0.3 = 1.5$

Measuring Cache Performance

- Ultimate metric is t_{avg}
 - Cache capacity and circuits roughly determines t_{hit}
 - Lower-level memory structures determine t_{miss}
 - Measure $\%_{miss}$
 - Hardware performance counters
 - Simulation

Capacity and Performance

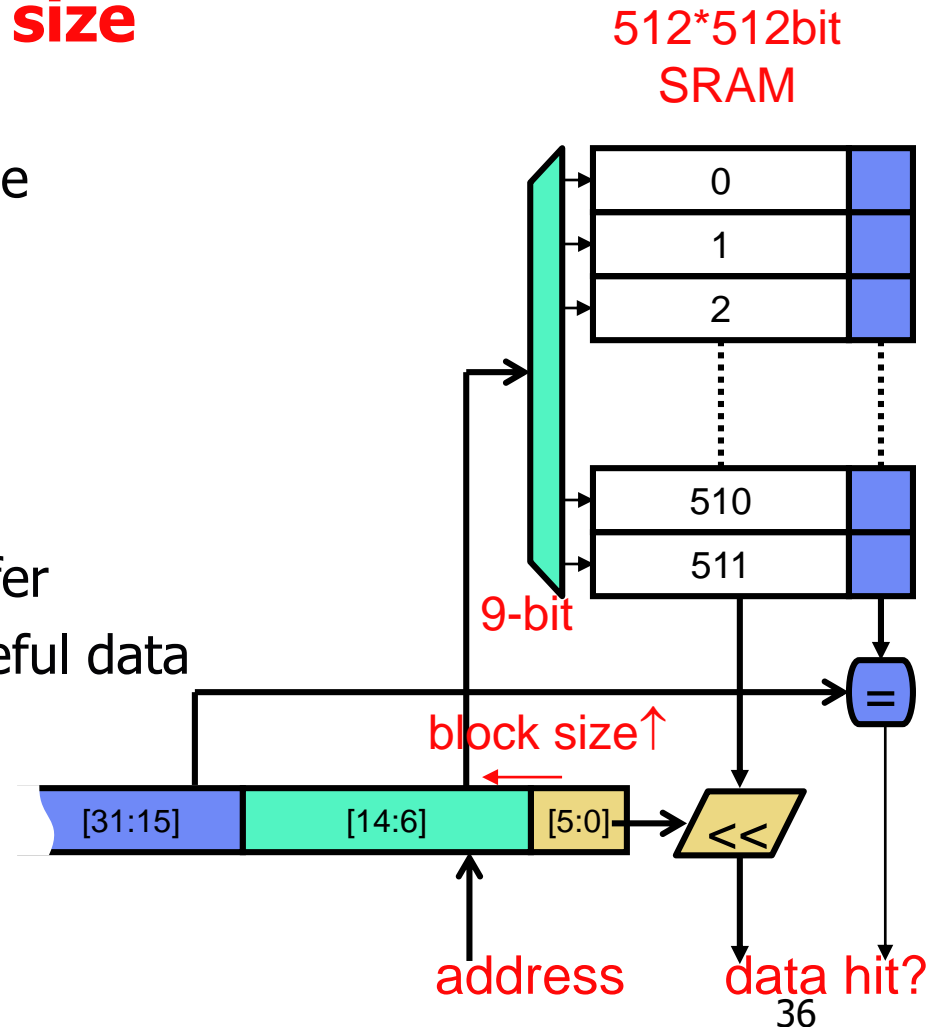
- Simplest way to reduce $\%_{\text{miss}}$: increase capacity
 - + Miss rate decreases monotonically
 - **“Working set”**: insns/data program is actively using
 - Diminishing returns
 - However t_{hit} increases
 - Latency grows with cache size
 - what happens to t_{avg} ?



- For fixed capacity, reduce $\%_{\text{miss}}$ by changing **organization**

Block Size

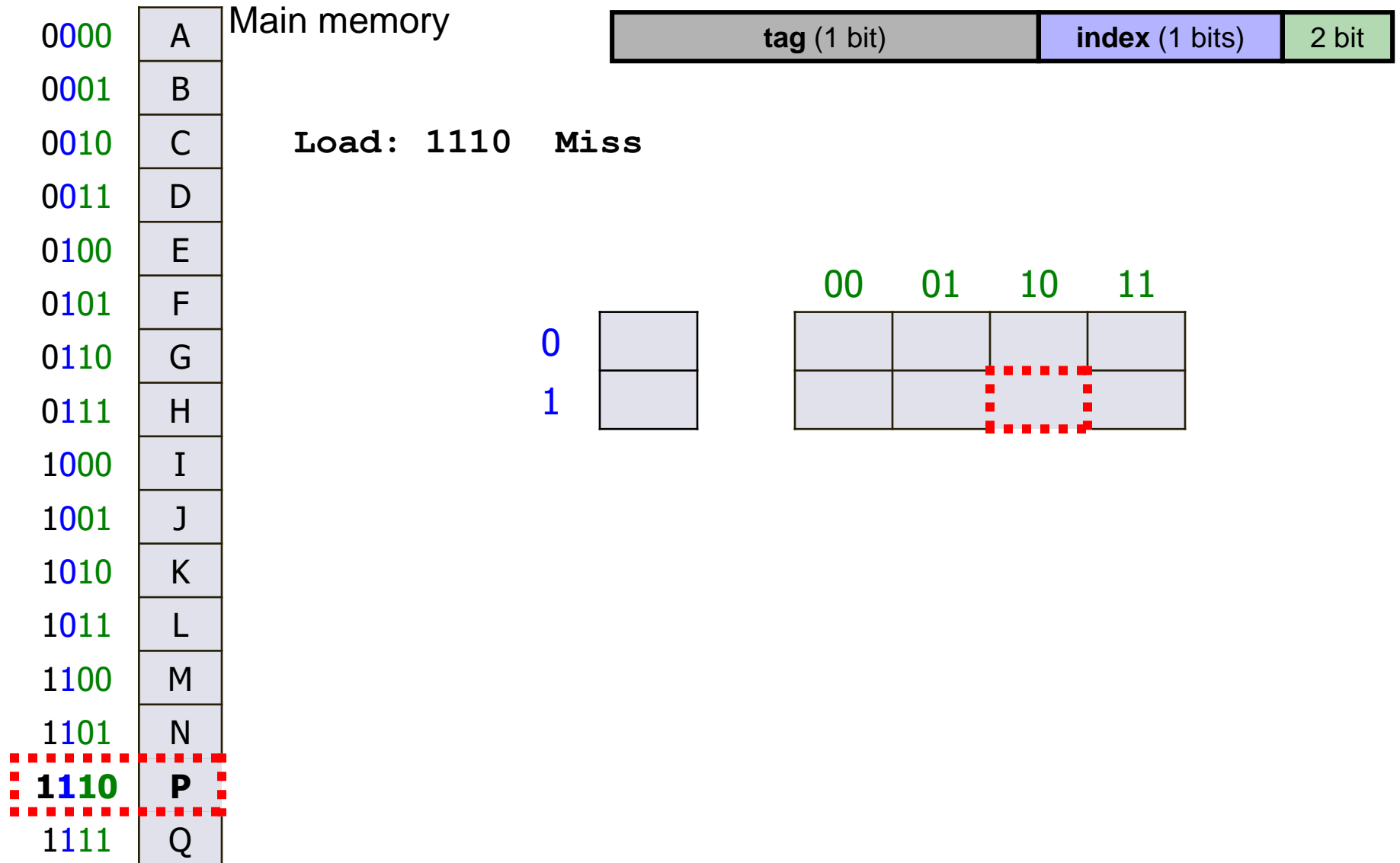
- Given capacity, manipulate $\%_{\text{miss}}$ by changing organization
- One option: increase **block size**
 - Exploit **spatial locality**
 - Notice index/offset bits change
 - Tag remains the same
- Ramifications
 - + Reduce $\%_{\text{miss}}$ (up to a point)
 - + Reduce tag overhead (why?)
 - Potentially useless data transfer
 - Premature replacement of useful data



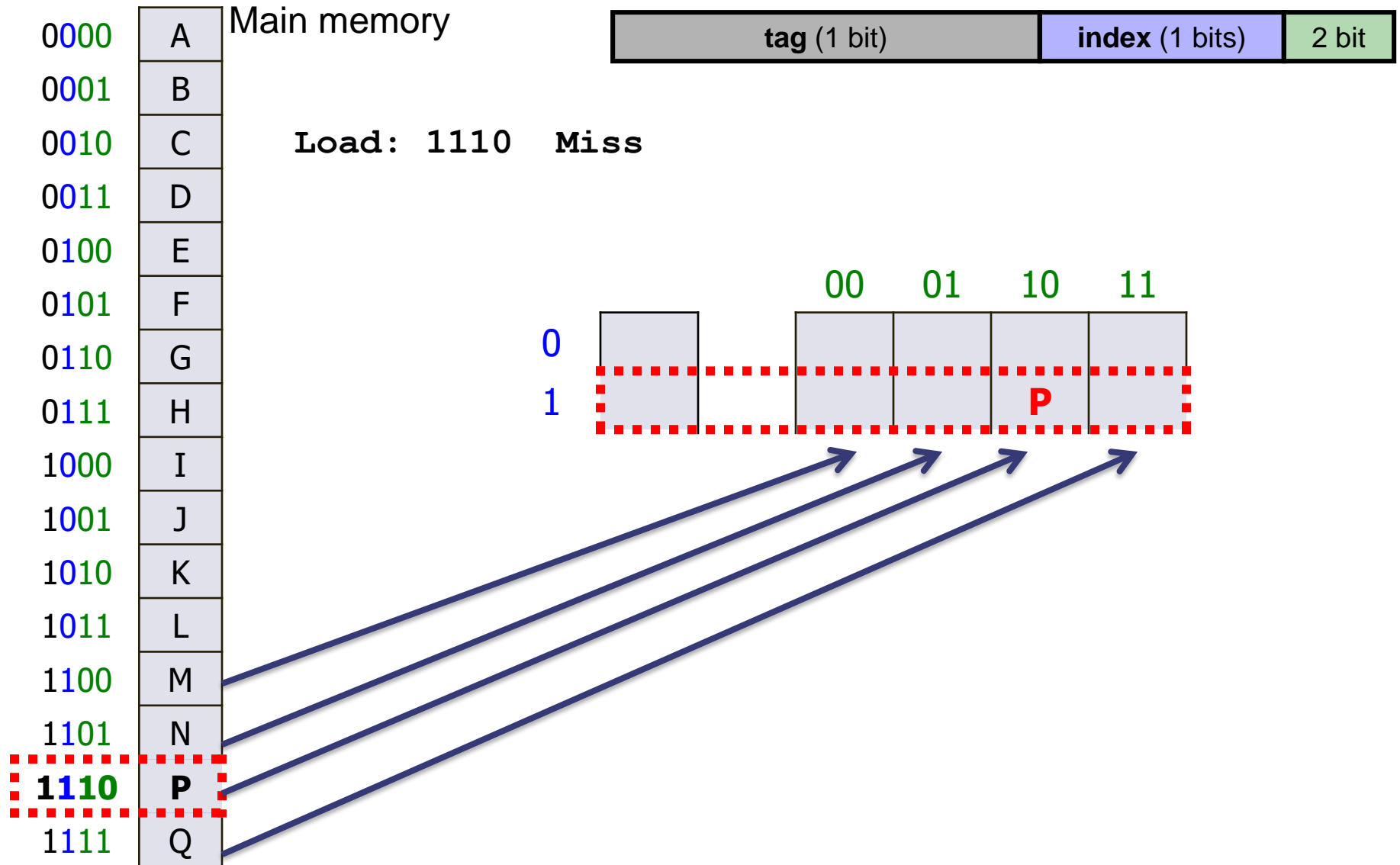
Larger Blocks to Lower Tag Overhead

- Tag overhead of 32KB cache with 1024 32B frames
 - 32B frames \rightarrow 5-bit offset
 - 1024 frames \rightarrow 10-bit index
 - 32-bit address $-$ 5-bit offset $-$ 10-bit index = 17-bit tag
 - (17-bit tag + 1-bit valid) * 1024 frames = 18Kb tags = 2.2KB tags
 - $\sim 6\%$ overhead
- Tag overhead of 32KB cache with 512 64B frames
 - 64B frames \rightarrow 6-bit offset
 - 512 frames \rightarrow 9-bit index
 - 32-bit address $-$ 6-bit offset $-$ 9-bit index = 17-bit tag
 - (17-bit tag + 1-bit valid) * 512 frames = 9Kb tags = 1.1KB tags
 - + $\sim 3\%$ overhead

4-bit Address, 8B Cache, 4B Blocks

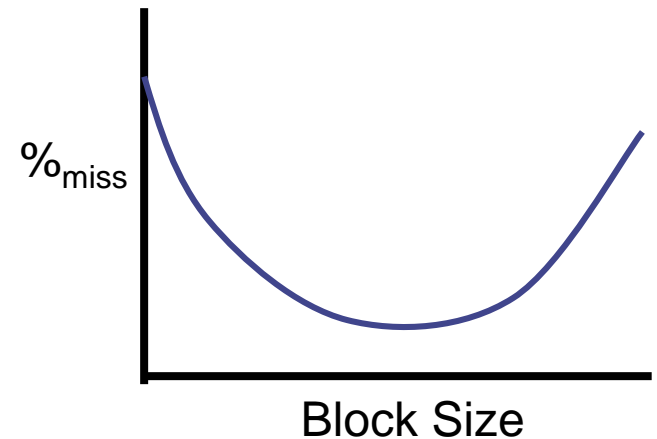


4-bit Address, 8B Cache, 4B Blocks



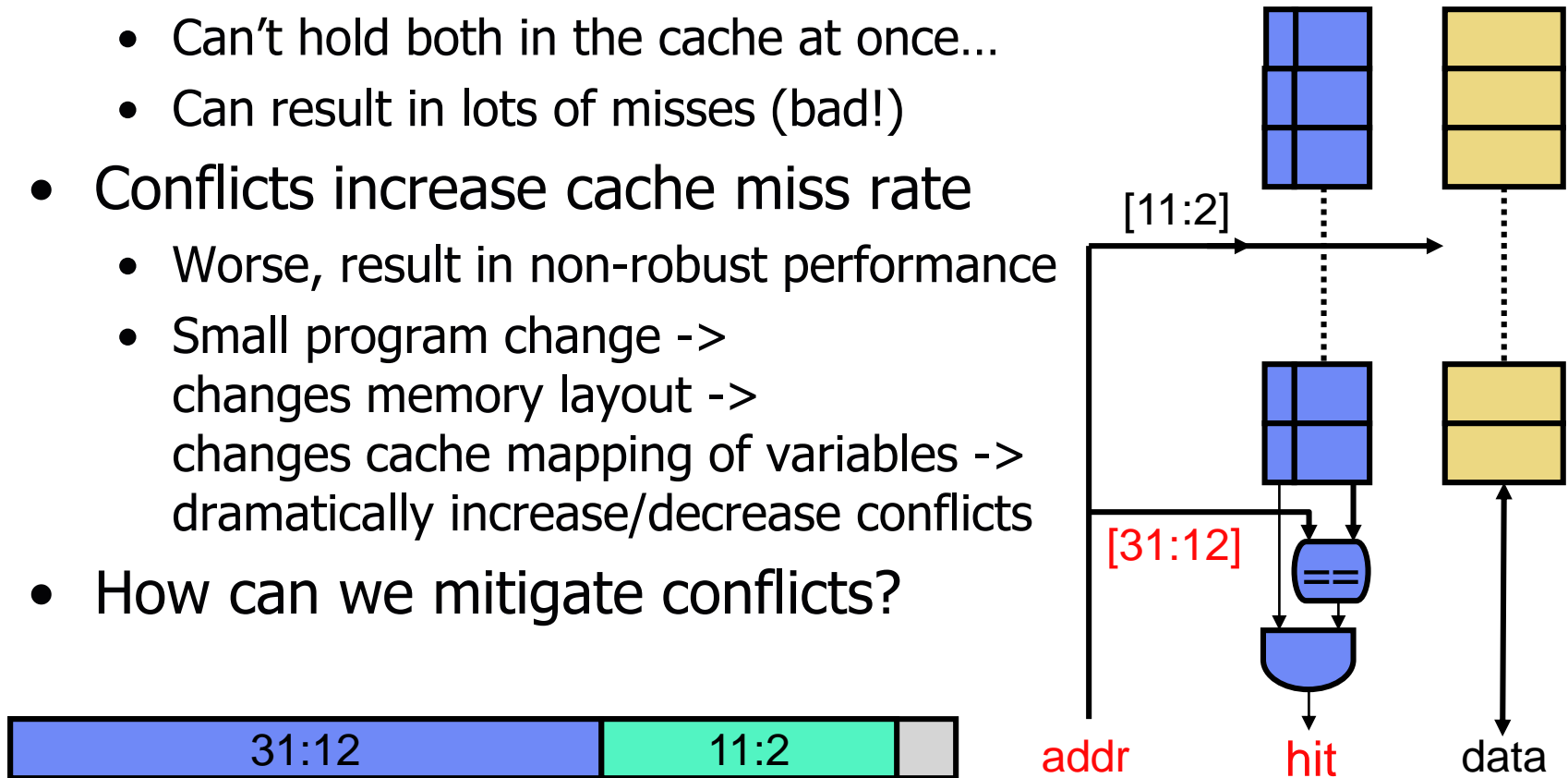
Effect of Block Size on Miss Rate

- Two effects on miss rate
 - + **Spatial prefetching (good)**
 - For adjacent locations
 - Turns miss/miss into miss/hit pairs
 - **Interference (bad)**
 - For non-adjacent locations that map to adjacent frames
 - Turns hits into misses by disallowing simultaneous residence
 - Consider entire cache as one big block
- Both effects always present
 - Spatial “prefetching” dominates initially
 - Depends on size of the cache
 - Reasonable block sizes are 32B–128B
- But also increases traffic
 - More data moved, not all used



Cache Conflicts

- Consider two frequently-accessed variables...
- What if their addresses have the same "index" bits?
 - Such addresses "conflict" in the cache
 - Can't hold both in the cache at once...
 - Can result in lots of misses (bad!)
- Conflicts increase cache miss rate
 - Worse, result in non-robust performance
 - Small program change -> changes memory layout -> changes cache mapping of variables -> dramatically increase/decrease conflicts
- How can we mitigate conflicts?



Associativity

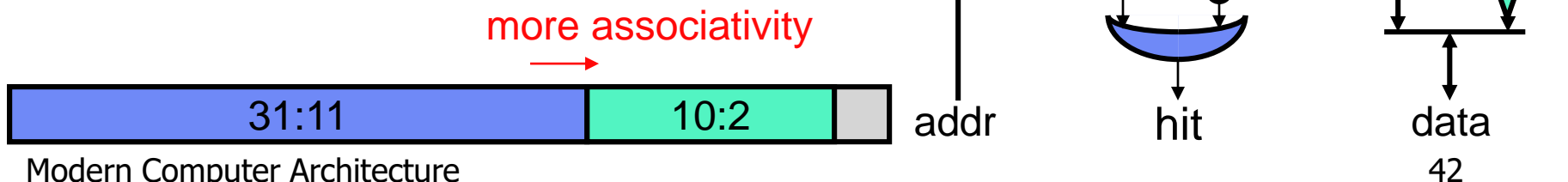
- **Set-associativity**

- Block can reside in one of few frames
- Frame groups called **sets**
- Each frame in a set called a **way**
- This is **2-way set-associative (SA)**
- 1-way → **direct-mapped (DM)**
- 1-set → **fully-associative (FA)**

+ Reduces conflicts

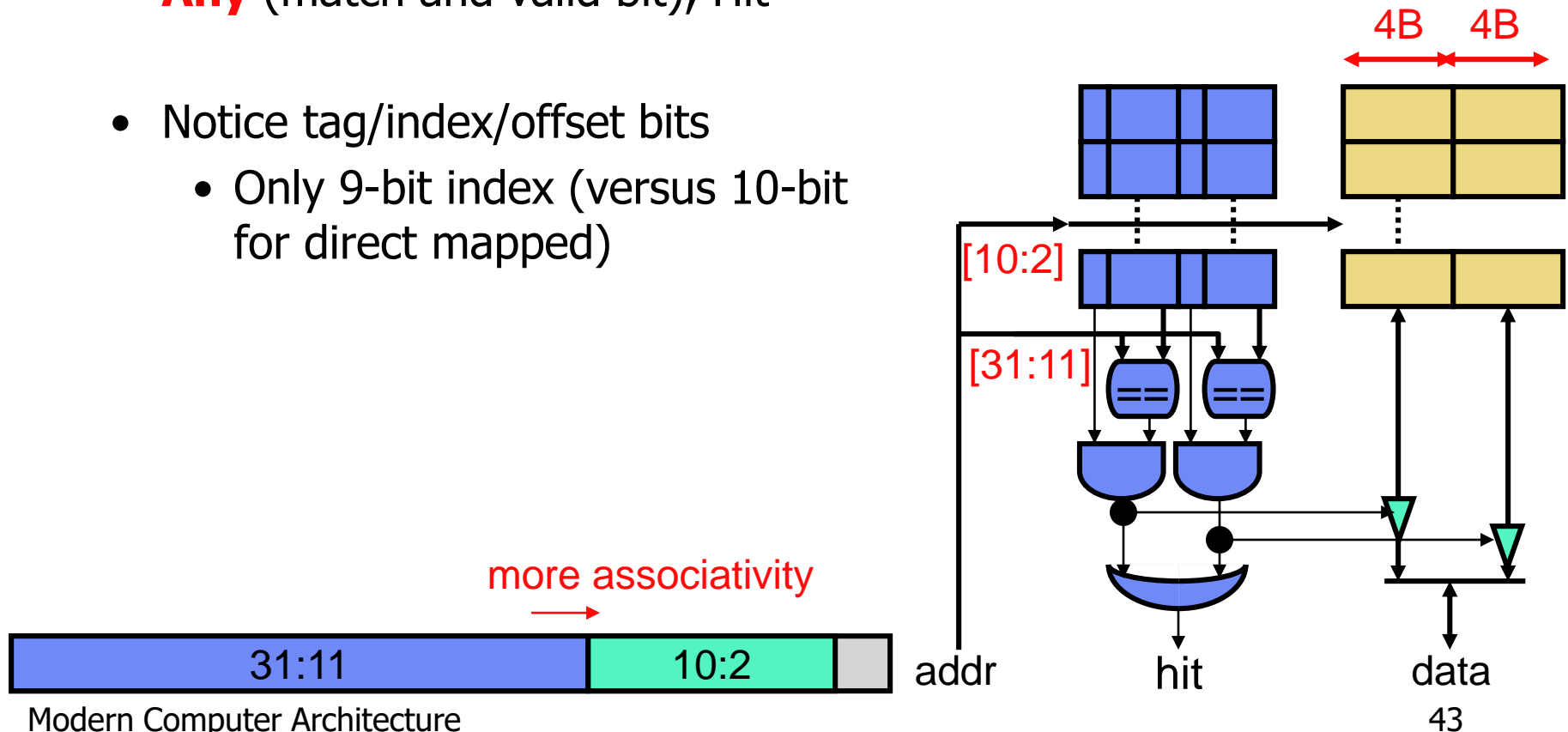
– Increases latency_{hit}:

- additional tag match & muxing



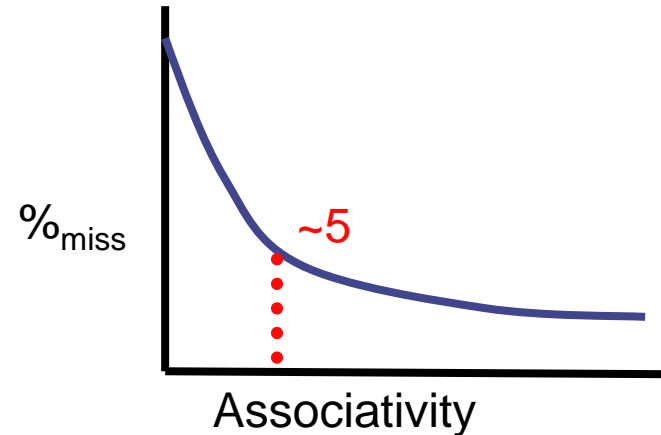
Associativity

- Lookup algorithm
 - Use index bits to find set
 - Read data/tags in all frames in parallel
 - **Any** (match and valid bit), Hit
- Notice tag/index/offset bits
 - Only 9-bit index (versus 10-bit for direct mapped)



Associativity and Performance

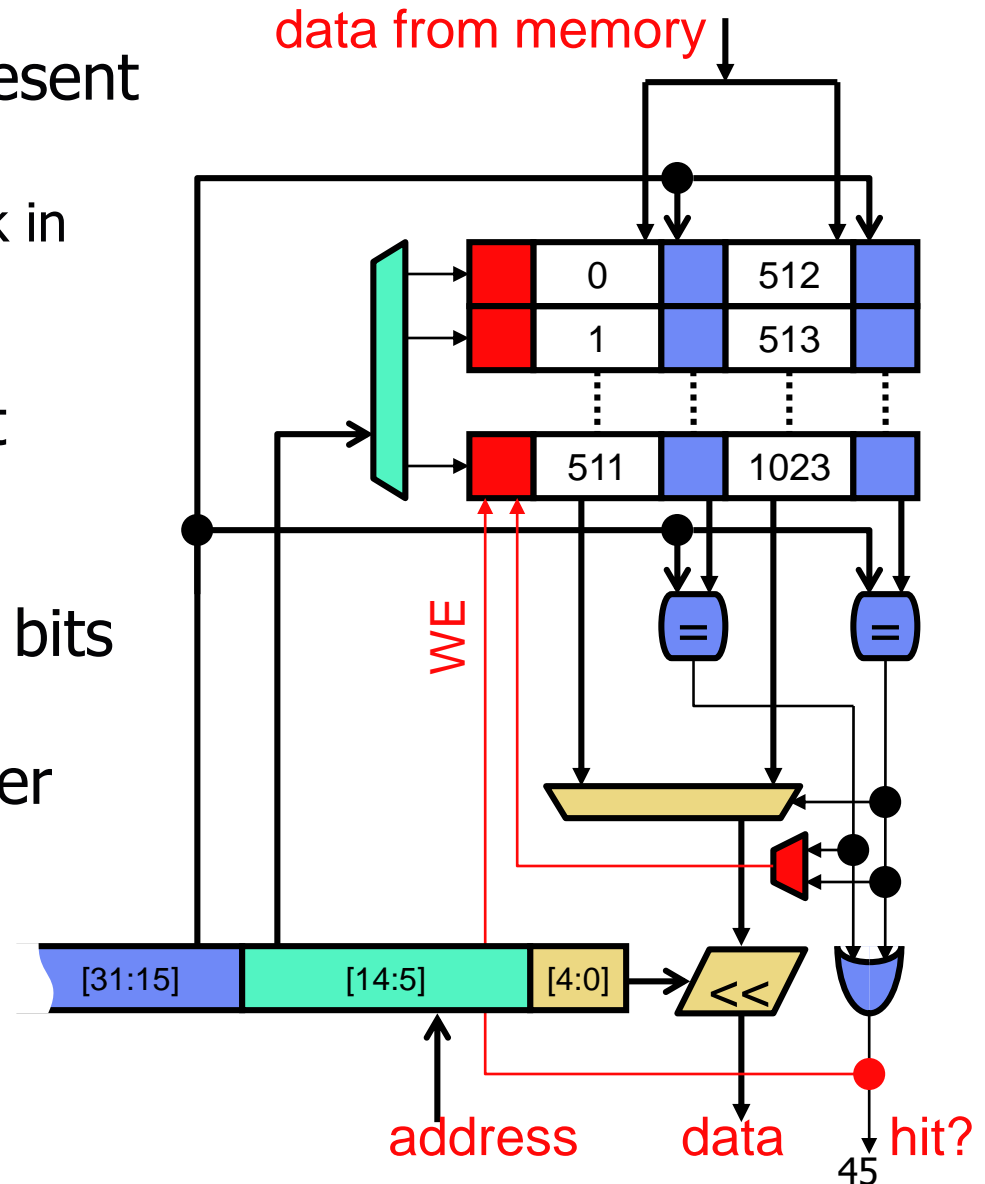
- Higher associative caches
 - + Have better (lower) $\%_{\text{miss}}$
 - Diminishing returns
 - However t_{hit} increases
 - The more associative, the slower
 - What about t_{avg} ?



- Block-size and number of sets should be powers of two
 - Makes indexing easier (just rip bits out of the address)
- 3-way set-associativity? No problem

Miss Handling & Replacement Policies

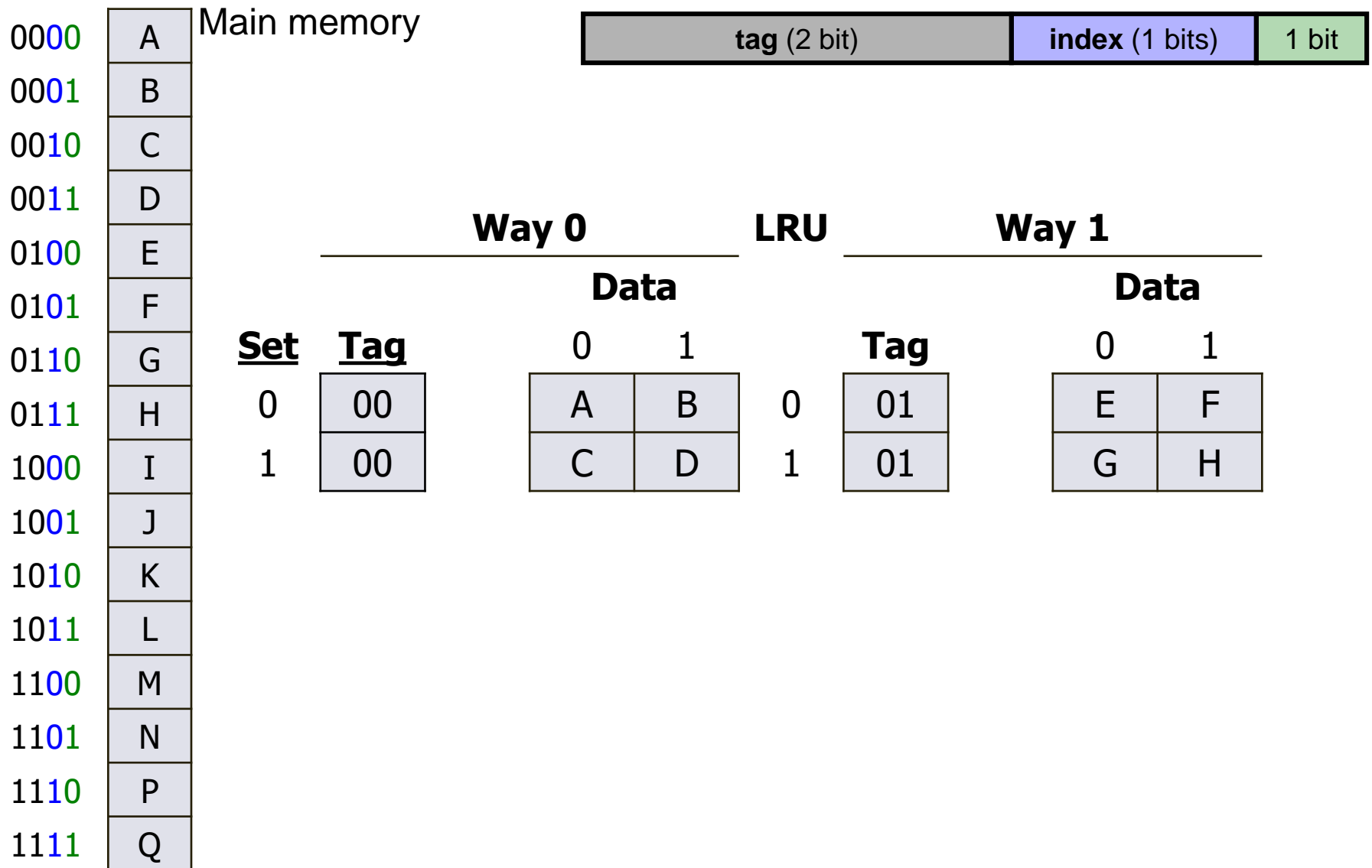
- Set-associative caches present a new design choice
 - On cache miss, which block in set to replace (kick out)?
- Add **LRU** field to each set
 - “Least recently used”
- Each access updates LRU bits
- Pseudo-LRU used for larger associativity caches



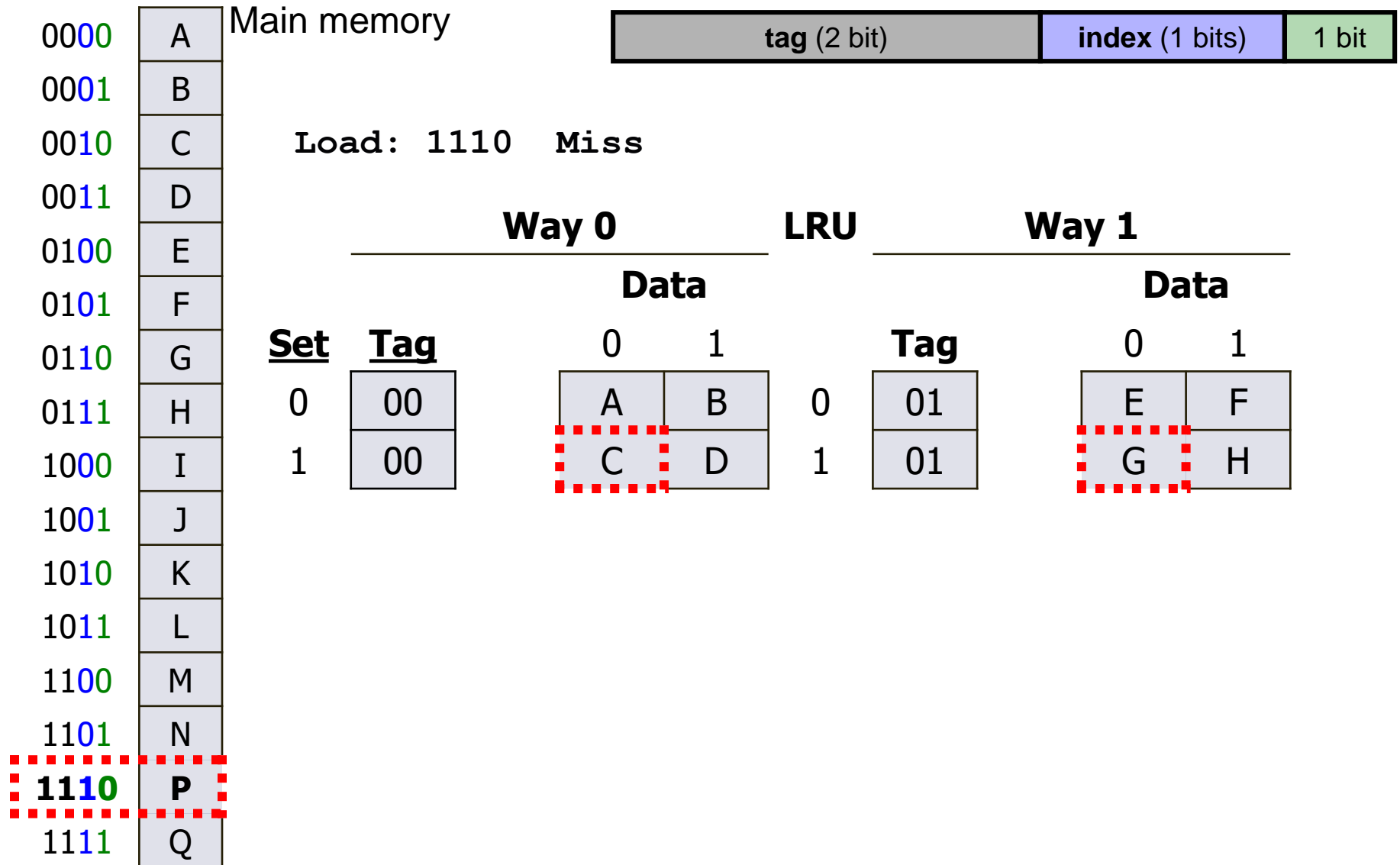
Replacement Policies

- Set-associative caches present a new design choice
 - On cache miss, which block in set to replace (kick out)?
- Some options
 - **Random**
 - **FIFO (first-in first-out)**
 - **LRU (least recently used)**
 - Fits with temporal locality, LRU = least likely to be used in future
 - **NMRU (not most recently used)**
 - An easier to implement approximation of LRU
 - Same as LRU for 2-way set-associative caches
 - **Belady's**: replace block that will be used furthest in future
 - Unachievable optimum

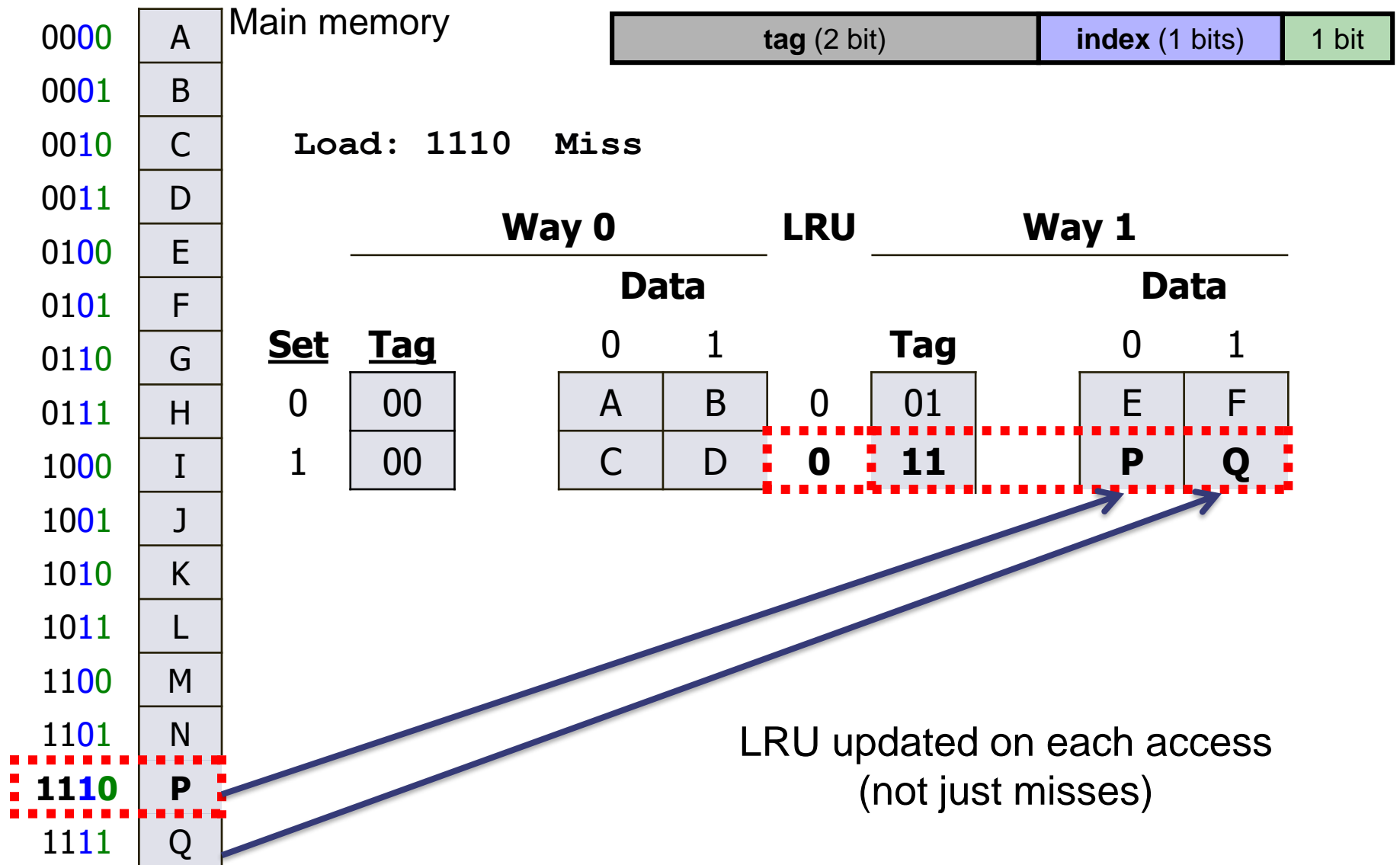
4-bit Address, 8B Cache, 2B Blocks, 2-way



4-bit Address, 8B Cache, 2B Blocks, 2-way



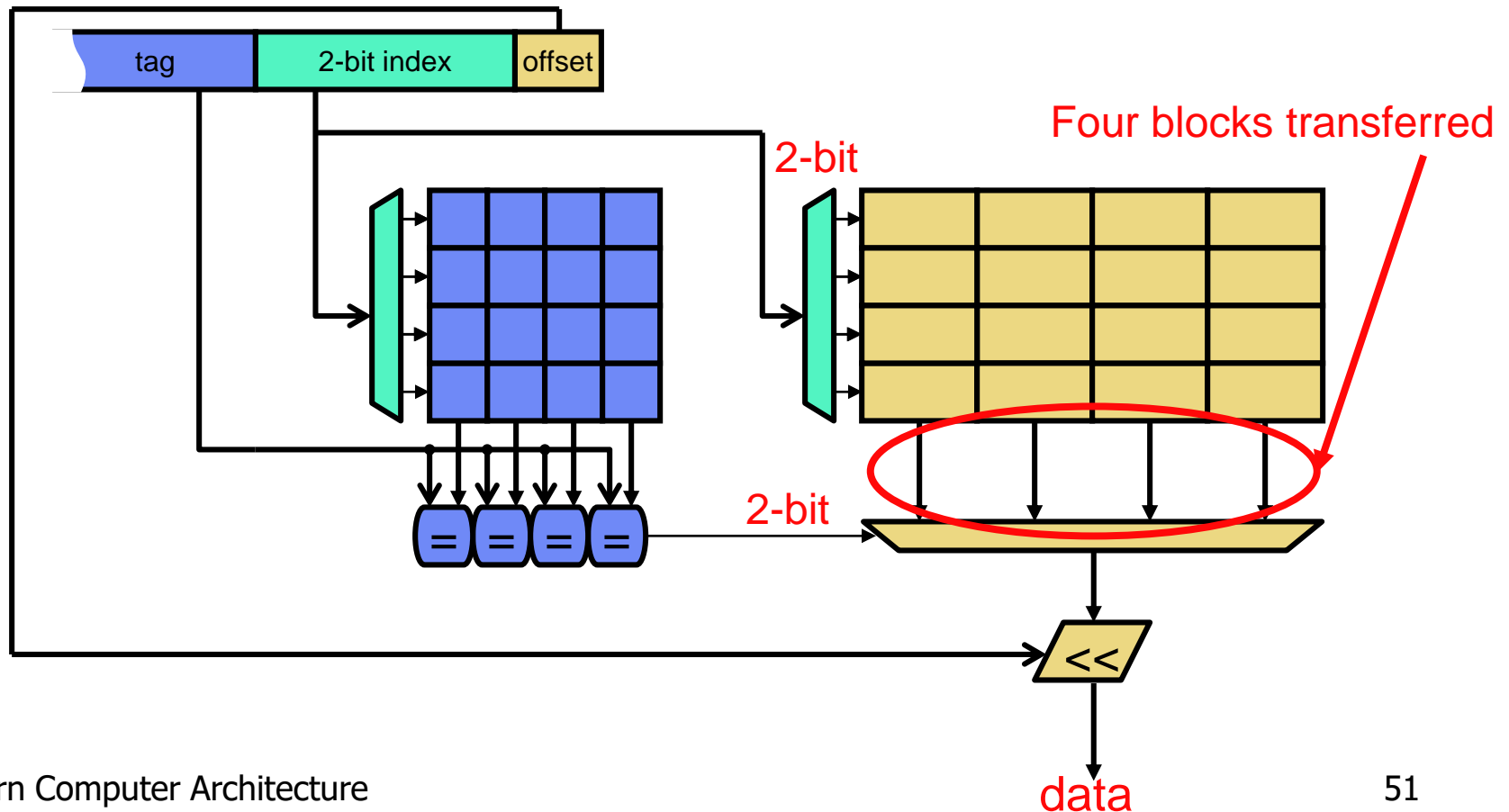
4-bit Address, 8B Cache, 2B Blocks, 2-way



Implementing Set-Associative Caches

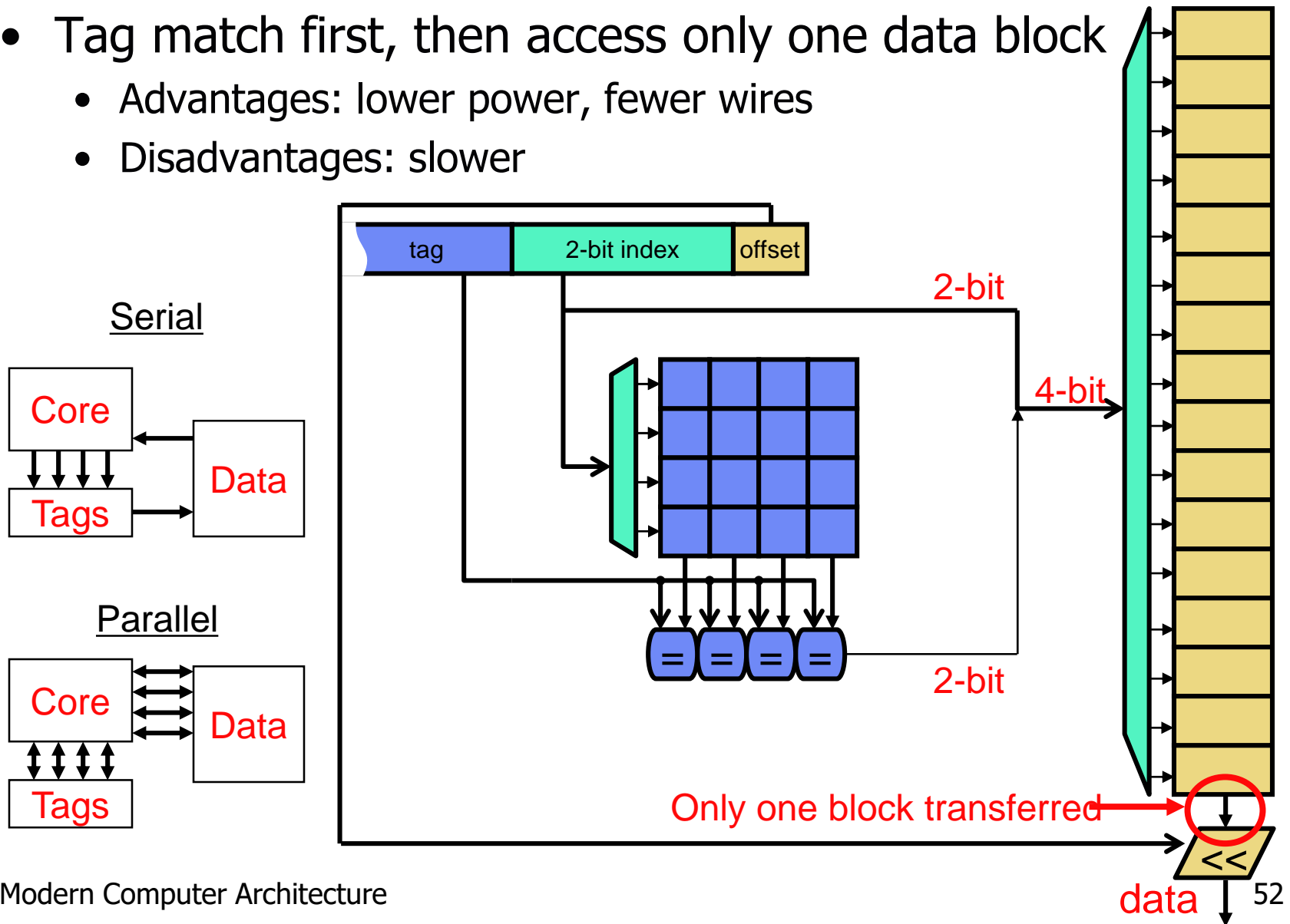
Option#1: Parallel Tag Access

- Data and tags actually physically separate
 - Split into two different memory structures
- Option#1: read both structures in parallel:



Option#2: Serial Tag Access

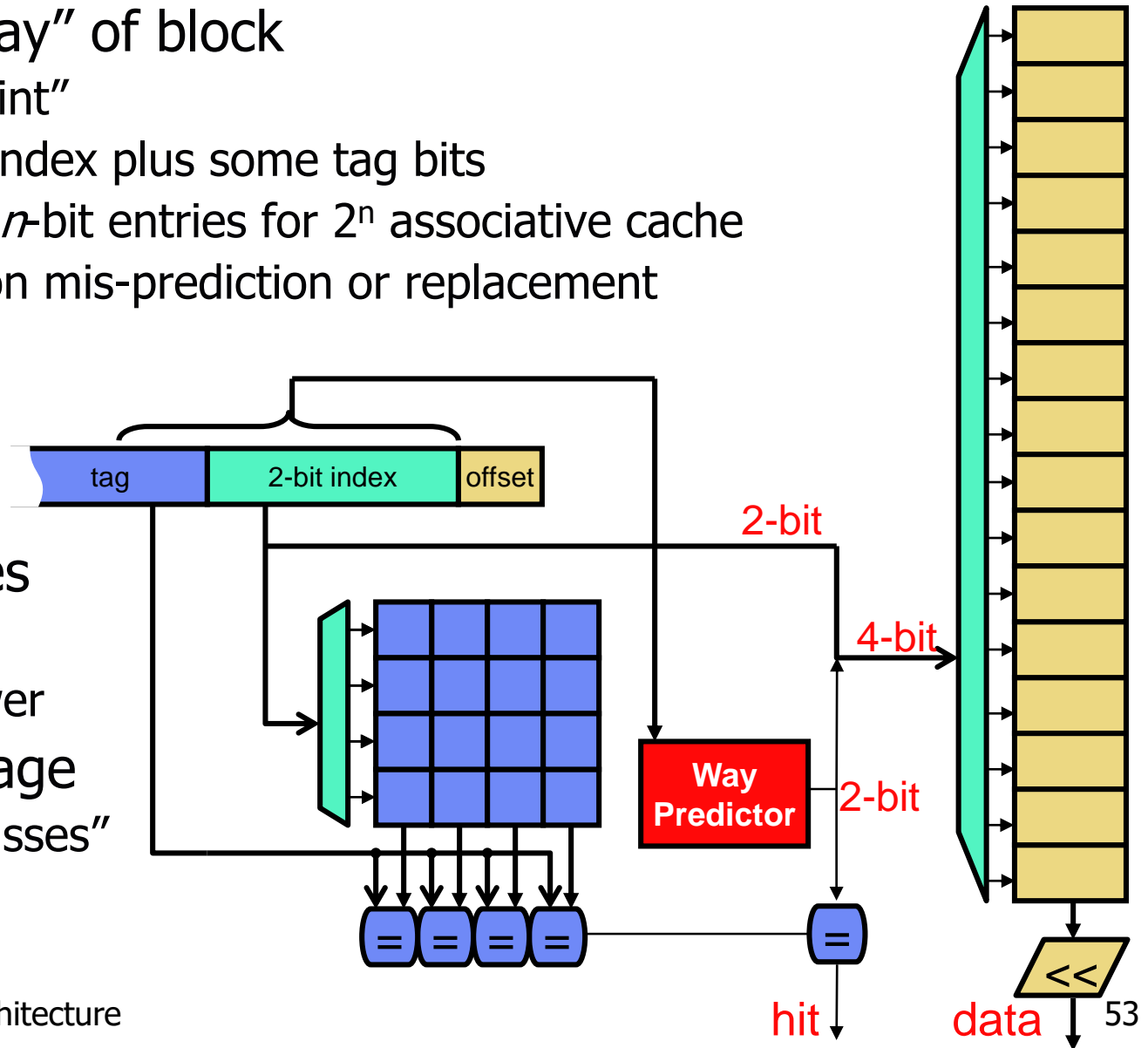
- Tag match first, then access only one data block
 - Advantages: lower power, fewer wires
 - Disadvantages: slower



Best of Both? Way Prediction

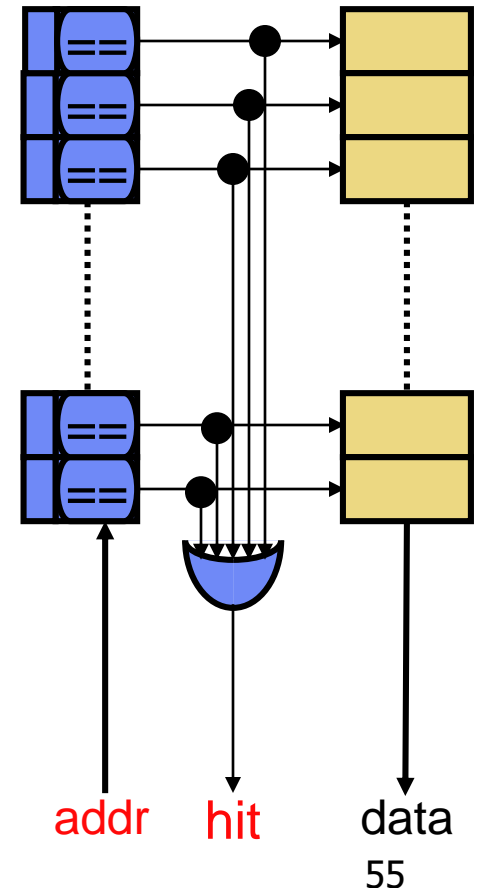
- Predict “way” of block
 - Just a “hint”
 - Use the index plus some tag bits
 - Table of n -bit entries for 2^n associative cache
 - Update on mis-prediction or replacement

- Advantages
 - Fast
 - Low-power
- Disadvantage
 - More “misses”



Highly Associative Caches with “CAMs”

- **CAM: content addressable memory**
 - Array of words with **built-in comparators**
 - **No separate “decoder” logic**
 - Input is value to match (tag)
 - Generates 1-hot encoding of matching slot
- Fully associative cache
 - Tags as CAM, data as RAM
 - Effective but somewhat expensive
 - But cheaper than any other method
 - Used for high (16-/32-way) associativity
 - No good way to build 1024-way associativity
 - + No real need for it, either
- CAMs are used elsewhere, too...



Cache Optimizations

Classifying Misses: 3C Model

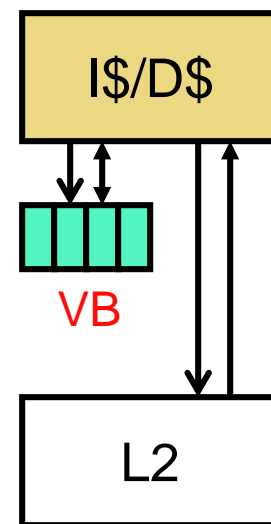
- Divide cache misses into three categories
 - **Compulsory (cold)**: never seen this address before
 - **Would miss even in infinite cache**
 - **Capacity**: miss because cache is too small
 - **Would miss even in fully associative cache**
 - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
 - **Conflict**: miss because cache associativity is too low
 - Identify? **All other misses**
 - **(Coherence)**: miss due to external invalidations
 - Only in shared memory multiprocessors (later)
- Calculated by multiple simulations
 - Simulate infinite cache, fully-associative cache, normal cache
 - Subtract to find each count

Miss Rate: ABC

- Why do we care about 3C miss model?
 - So that we know what to do to eliminate misses
 - e.g. if have no conflict misses, increasing associativity won't help
- **Associativity**
 - + Decreases conflict misses
 - Increases latency_{hit}
- **Block size**
 - Increases conflict/capacity misses
 - + Decreases compulsory misses
 - No significant effect on latency_{hit}
- **Capacity**
 - + Decreases capacity misses
 - Increases latency_{hit}

Reducing Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
 - High-associativity is expensive, but also rarely needed
 - 3 blocks mapping to same 2-way set
- **Victim buffer (VB)**: small fully-associative cache
 - Sits on I\$/D\$ miss path
 - Small so very fast (e.g., 8 entries)
 - Blocks evicted from I\$/D\$ are placed in VB
 - On miss, check VB: hit? Place block back in I\$/D\$
 - 8 extra ways, shared among all sets
 - + Only a few sets will need it at any given time
 - + Very effective in practice

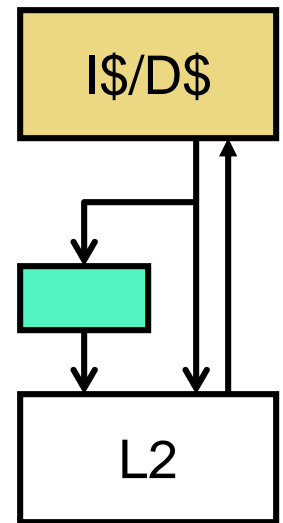


Overlapping Misses: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
 - Consider: Load [r1] -> r2; Load [r3] -> r4; Add r2, r4 -> r5
- **Handle misses in parallel**
 - Allows “overlapping” misses
 - “memory-level parallelism”
- Implementation: **miss status holding register (MSHR)**
 - Remember: miss address, chosen frame, requesting instruction
 - When miss returns know where to put block, who to inform

Prefetching

- Bring data into cache proactively/**speculatively**
 - If successful, reduces number of cache misses
- Key: anticipate upcoming miss addresses accurately
 - Can do in software or hardware
- Simple hardware prefetching: **next block prefetching**
 - Miss on address **X** → anticipate miss on **X+block-size**
 - + Works for insns: sequential execution
 - + Works for data: arrays
- Table-driven hardware prefetching
 - Use **predictor** to detect strides, common patterns
- Effectiveness determined by:
 - **Timeliness**: initiate prefetches sufficiently in advance
 - **Coverage**: prefetch for as many misses as possible
 - **Accuracy**: don't pollute with unnecessary data



Software Prefetching

- Use a special “prefetch” instruction
 - Tells the hardware to bring in data, doesn’t actually read it
 - Just a hint
- Inserted by programmer or compiler

- Example

```
int tree_add(tree_t* t) {  
    if (t == NULL) return 0;  
    __builtin_prefetch(t->left);  
    return t->val + tree_add(t->right) + tree_add(t->left);  
}
```

- Multiple prefetches bring multiple blocks in parallel
 - More “memory-level” parallelism (MLP)

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
 - Several code restructuring techniques to improve both
 - Loop blocking (break into cache-sized chunks), loop fusion
 - Compiler must know that restructuring preserves semantics
- **Loop interchange**: spatial locality
 - Example: for a row-major matrix, `x[i][j]` should be followed by `x[i][j+1]`
 - Poor code: `x[i][j]` followed by `x[i+1][j]`

```
for (j = 0; j < NCOLS; j++)  
    for (i = 0; i < NROWS; i++)  
        sum += X[i][j];
```
 - Better code

```
for (i = 0; i < NROWS; i++)  
    for (j = 0; j < NCOLS; j++)  
        sum += X[i][j];
```

Software Restructuring: Data

- **Loop blocking**: temporal locality

- Poor code

```
for (k=0; k<NUM_ITERATIONS; k++)  
    for (i=0; i<NUM_ELEMS; i++)  
        x[i] = f(x[i]);    // say
```

- Better code

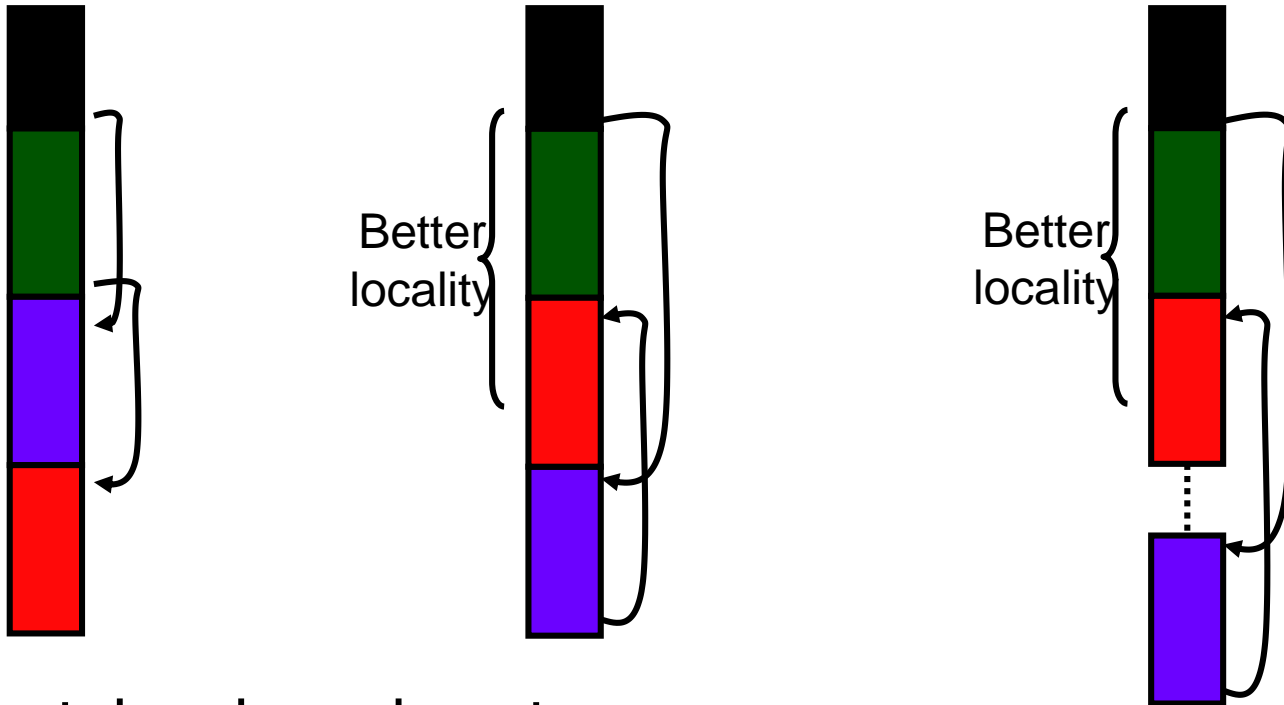
- Cut array into CACHE_SIZE chunks
 - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NUM_ELEMS; i+=CACHE_SIZE)  
    for (k=0; k<NUM_ITERATIONS; k++)  
        for (j=0; j<CACHE_SIZE; j++)  
            x[i+j] = f(x[i+j]);
```

- Assumes you know CACHE_SIZE, do you?

Software Restructuring: Code

- Compiler can layout code for better locality
 - If (a) { **code1;** } else { **code2;** } **code3;**
 - But, code2 case never happens (say, error condition)

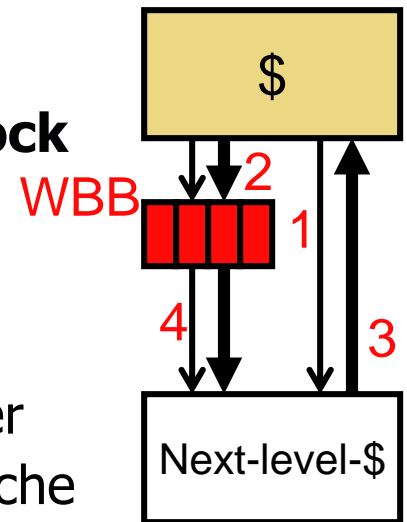


- Fewer taken branches, too

What About Stores?

Handling Cache Writes

- When to propagate new value to (lower level) memory?
 - **Option #1: Write-through**: immediately
 - On hit, update cache
 - Immediately send the write to the next level
 - **Option #2: Write-back**: when block is replaced
 - Requires additional **“dirty”** bit per block
 - Replace **clean** block: **no extra traffic**
 - Replace **dirty** block: **extra “writeback” of block**
- + **Writeback-buffer (WBB)**:
- Hide latency of writeback (keep off critical path)
 - Step#1: Send “fill” request to next-level
 - Step#2: While waiting, write dirty block to buffer
 - Step#3: When new blocks arrives, put it into cache
 - Step#4: Write buffer contents to next-level



Determining WBB Size

- How big should we make the L1's WBB?
- $L = \lambda W$ (Little's Law)
- $L = (\%miss_{L1}) * (latency_{L2})$
- configuration:
 - 1GHz processor
 - 20% of insns are loads
 - $\%miss_{L1} = 5\%$
 - L2 access = 20 cycles
 - $L = .01 * 20 \rightarrow 1$ entry suffices

Write Propagation Comparison

- **Write-through**

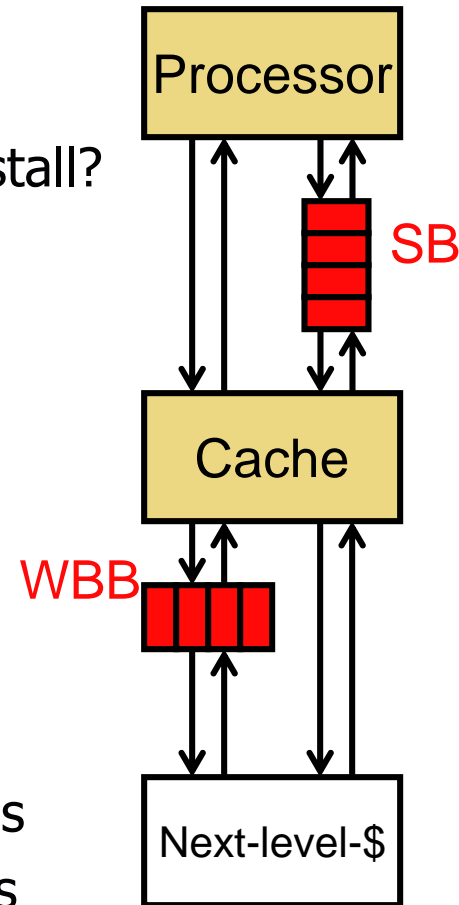
- Creates additional traffic
 - Consider repeated write hits
- Next level must handle small writes (1, 2, 4, 8-bytes)
- + No need for dirty bits in cache
- + No need to handle “writeback” operations
 - Simplifies miss handling (no write-back buffer)
- Sometimes used for L1 caches (IBM, GPUs)
- Usually **write-non-allocate**: on write miss, just write to next level

- **Write-back**

- + Key advantage: uses less bandwidth
- Reverse of other pros/cons above
- Used by Intel, AMD, and many ARM cores
- Second-level and beyond are generally write-back caches
- Usually **write-allocate**: on write miss, fill block from next level

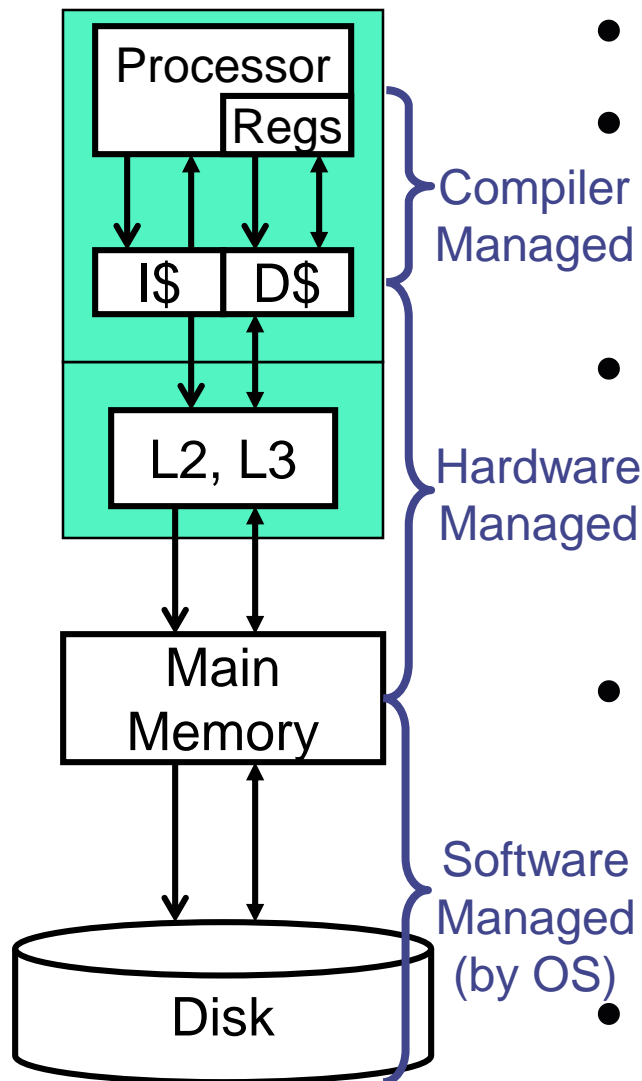
Write Misses and Store Buffers

- Read miss?
 - Load can't go on without the data, it must stall
- Write miss?
 - Technically, no instruction is waiting for data, why stall?
- **Store buffer**: a small buffer for store misses
 - Stores put address/value into SB, **keep going**
 - SB writes to D\$ in the background
 - Loads must search SB (in addition to D\$)
 - (mostly) eliminates stalls on write misses
 - creates some problems in multiprocessors (later)
- Store buffer vs. writeback buffer
 - Store buffer: "in front" of D\$, for hiding store misses
 - Writeback buffer: "behind" D\$, for hiding writebacks



Cache Hierarchies

Concrete Memory Hierarchy



- 0th level: **Registers**
- 1st level: **Primary caches**
 - Split instruction (I\$) and data (D\$)
 - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
 - On-chip, typically made of SRAM
 - 2nd level typically ~256KB to 512KB
 - “Last level cache” typically 4MB to 16MB
- 3rd level: **main memory**
 - Made of DRAM (“Dynamic” RAM)
 - Typically 1GB to 4GB for desktops/laptops
 - Servers can have >1 TB
- 4th level: **disk (swap and files)**
 - Uses magnetic disks or flash drives

Designing a Cache Hierarchy

- For any memory component: t_{hit} vs. $\%_{\text{miss}}$ tradeoff
- Upper components (I\$, D\$) emphasize low t_{hit}
 - **Frequent access** → t_{hit} **important**
 - t_{miss} is low → $\%_{\text{miss}}$ less important
 - Lower capacity and lower associativity (to reduce t_{hit})
 - Small-medium block-size (to reduce conflicts)
 - **Split instruction & data cache to allow simultaneous access**
- Moving down (L2, L3) emphasis turns to $\%_{\text{miss}}$
 - **Infrequent access** → t_{hit} **less important**
 - t_{miss} is high → $\%_{\text{miss}}$ important
 - High capacity, associativity, and block size (to reduce $\%_{\text{miss}}$)
 - **Unified insn+data caches to dynamically allocate capacity**

Split vs. Unified Caches

- **Split I\$/D\$**: insns and data in different caches
 - To minimize structural hazards and t_{hit}
 - Larger unified I\$/D\$ would be slow, 2nd port even slower
 - Optimize I\$ to fetch multiple instructions, no writes
- **Unified L2, L3**: insns and data together
 - To minimize $\%_{\text{miss}}$
 - + Fewer capacity misses: unused insn capacity can be used for data
 - More conflict misses: insn/data conflicts
 - A much smaller effect in large caches
 - Insn/data structural hazards are rare: simultaneous I\$/D\$ miss
 - Go even further: unify L2/L3 of multiple cores in a multi-core

Hierarchy: Inclusion versus Exclusion

- **Inclusion**

- **Bring block from memory into L2 then L1**
 - A block in the L1 is always in the L2 (but not vice-versa)
- If block evicted from L2, must also evict it from L1
 - Why? more on this when we talk about multicore

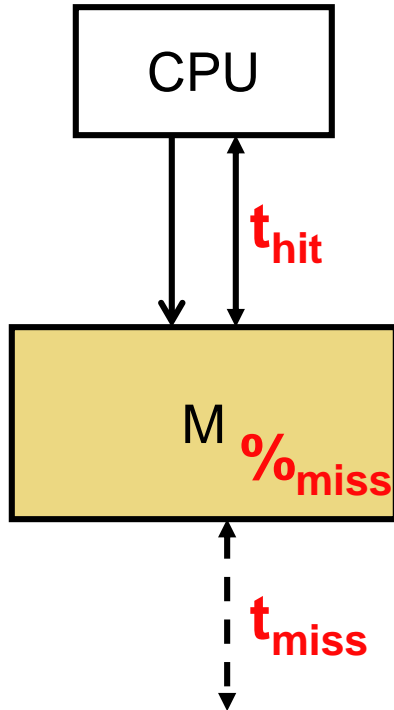
- **Exclusion**

- **Bring block from memory into L1 but not L2**
 - Move block to L2 on L1 eviction
 - L2 becomes a large victim cache
 - Block is either in L1 or L2 (never both)
- Good if L2 is small relative to L1
 - Example: AMD's Duron 64KB L1s, 64KB L2

- **Non-inclusion**

- No guarantees

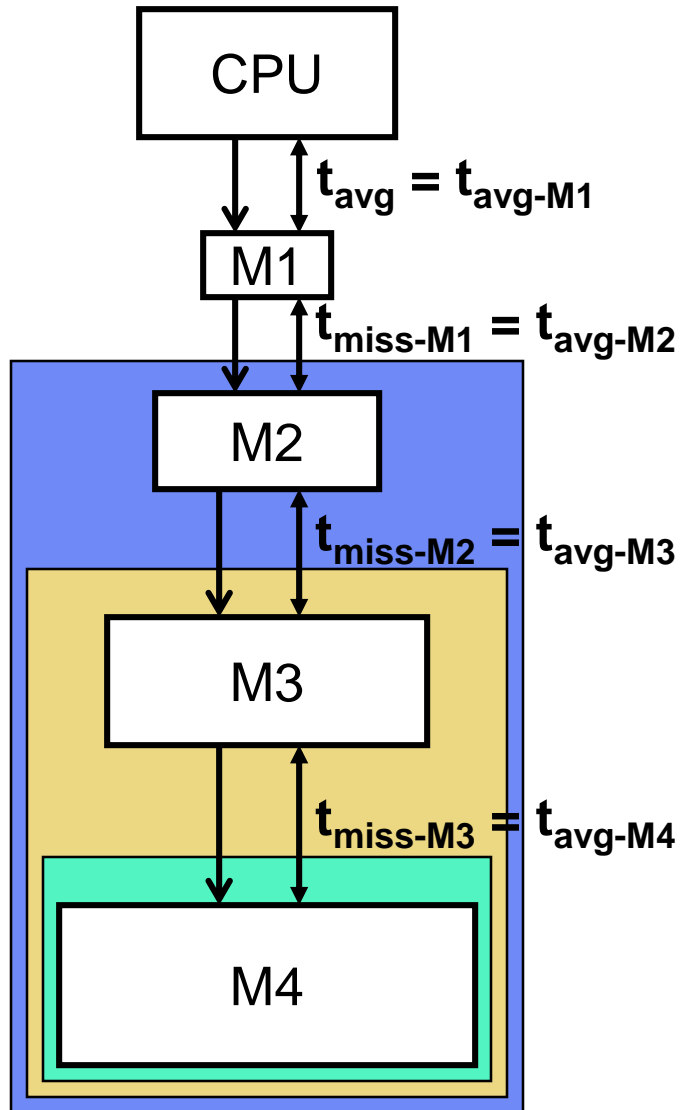
Memory Performance Equation



- For memory component M
 - **Access**: read or write to M
 - **Hit**: desired data found in M
 - **Miss**: desired data not found in M
 - Must get from another (slower) component
 - **Fill**: action of placing data in M
- $\%_{miss}$ (miss-rate): $\#misses / \#accesses$
- t_{hit} : time to read data from (write data to) M
- t_{miss} : time to read data into M
- Performance metric
 - t_{avg} : average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Hierarchy Performance



$$\begin{aligned}
 &t_{avg} \\
 &t_{avg-M1} \\
 &t_{hit-M1} + (\%_{miss-M1} * t_{miss-M1}) \\
 &t_{hit-M1} + (\%_{miss-M1} * t_{avg-M2}) \\
 &t_{hit-M1} + (\%_{miss-M1} * (t_{hit-M2} + (\%_{miss-M2} * t_{miss-M2}))) \\
 &t_{hit-M1} + (\%_{miss-M1} * (t_{hit-M2} + (\%_{miss-M2} * t_{avg-M3}))) \\
 &\dots
 \end{aligned}$$

Performance Calculation

- In a pipelined processor, I\$/D\$ t_{hit} is “built in” (effectively 0)
- Parameters
 - Base pipeline CPI = 1
 - Instruction mix: 30% loads/stores
 - I\$: $\%_{miss} = 2\%$, $t_{miss} = 10$ cycles
 - D\$: $\%_{miss} = 10\%$, $t_{miss} = 10$ cycles
- What is CPI with cache misses?
 - $CPI_{I\$} = \%_{missI\$} * t_{miss} = 0.02 * 10 \text{ cycles} = 0.2 \text{ cycle}$
 - $CPI_{D\$} = \%_{memory} * \%_{missD\$} * t_{missD\$} = 0.30 * 0.10 * 10 \text{ cycles} = 0.3 \text{ cycle}$
 - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$} = 1 + 0.2 + 0.3 = 1.5$

Miss Rates: per “access” vs “instruction”

- Miss rates can be expressed two ways:
 - Misses per “instruction” (or instructions per miss), -or-
 - Misses per “cache access” (or accesses per miss)
- For first-level caches, use instruction mix to convert
 - If memory ops are $1/3^{\text{rd}}$ of instructions..
 - 2% of instructions miss (1 in 50) is 6% of “accesses” miss (1 in 17)
- What about second-level caches?
 - Misses per “instruction” still straight-forward (“global” miss rate)
 - Misses per “access” is trickier (“local” miss rate)
 - Depends on number of accesses (which depends on L1 miss rate)

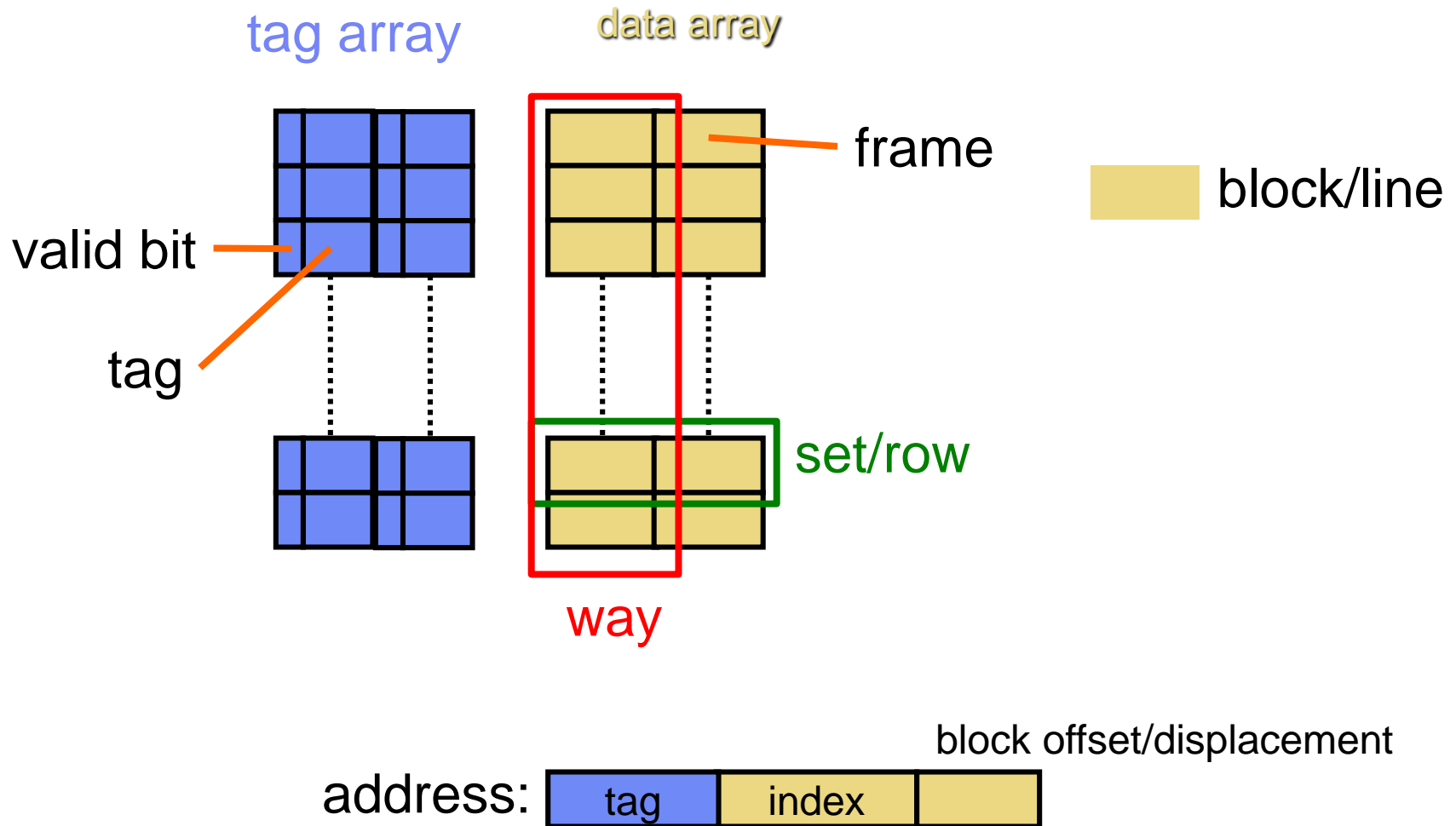
Performance Calculation (Revisited)

- Parameters
 - Base pipeline CPI = 1
 - In this case, already incorporates t_{hit}
 - I\$: $\%_{\text{miss}} = 2\%$ of instructions, $t_{\text{miss}} = 10$ cycles
 - D\$: $\%_{\text{miss}} = 3\%$ of instructions, $t_{\text{miss}} = 10$ cycles
- What is new CPI?
 - $\text{CPI}_{\text{I\$}} = \%_{\text{missI\$}} * t_{\text{miss}} = 0.02 * 10 \text{ cycles} = 0.2 \text{ cycle}$
 - $\text{CPI}_{\text{D\$}} = \%_{\text{missD\$}} * t_{\text{missD\$}} = 0.03 * 10 \text{ cycles} = 0.3 \text{ cycle}$
 - $\text{CPI}_{\text{new}} = \text{CPI} + \text{CPI}_{\text{I\$}} + \text{CPI}_{\text{D\$}} = 1 + 0.2 + 0.3 = 1.5$

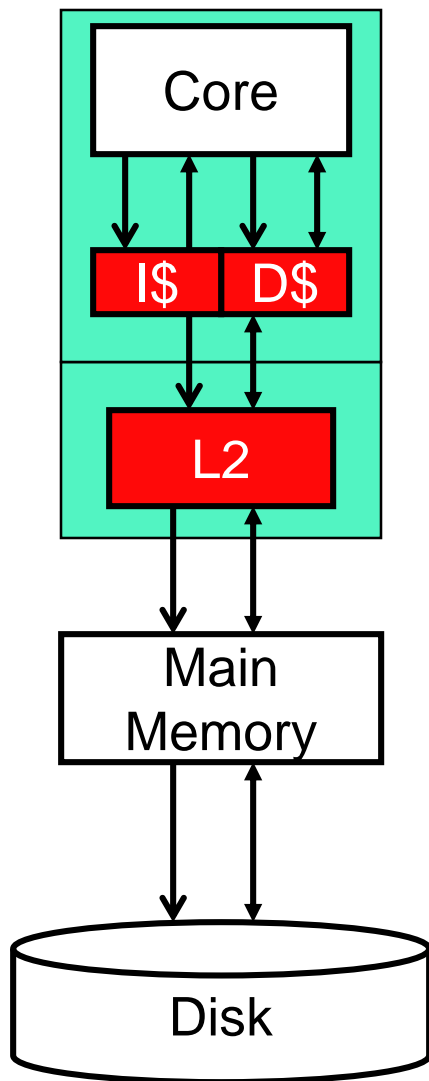
Multilevel Performance Calculation

- Parameters
 - 30% of instructions are memory operations
 - L1: $t_{\text{hit}} = 1$ cycles (included in CPI of 1), $\%_{\text{miss}} = 5\%$ of accesses
 - L2: $t_{\text{hit}} = 10$ cycles, $\%_{\text{miss}} = 20\%$ **of L2 accesses**
 - Main memory: $t_{\text{hit}} = 50$ cycles
- Calculate CPI
 - $\text{CPI} = 1 + 30\% * 5\% * t_{\text{missD\$}}$
 - $t_{\text{missD\$}} = t_{\text{avgL2}} = t_{\text{hitL2}} + (\%_{\text{missL2}} * t_{\text{hitMem}}) = 10 + (20\% * 50) = 20$ cycles
 - Thus, $\text{CPI} = 1 + 30\% * 5\% * 20 = 1.3$ CPI
- Alternate CPI calculation:
 - What % of instructions miss in L1 cache? $30\% * 5\% = 1.5\%$
 - What % of instructions miss in L2 cache? $20\% * 1.5\% = 0.3\%$ of insn
 - $\text{CPI} = 1 + (1.5\% * 10) + (0.3\% * 50) = 1 + 0.15 + 0.15 = 1.3$ CPI

Cache Glossary



Summary



- “Cache”: hardware managed
 - Hardware automatically retrieves missing data
 - Built from fast on-chip SRAM
 - In contrast to off-chip, DRAM “main memory”
- **Average access time** of a memory component
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure
→ memory hierarchy
- Cache ABCs (**associativity, block size, capacity**)
- **Performance optimizations**
 - Prefetching & data restructuring
- **Handling writes**
 - Write-back vs. write-through
- **Memory hierarchy**
 - Smaller, faster, expensive → bigger, slower, cheaper