CIS 5710
Computer Organization and Design

Unit 8: Caches

Slides by Profs. Amir Roth, Milo Martin,
C.J. Taylor and Benedict Brown

Readings

- P&H Chapter 5
  - 5.1-5.3, 5.5
- Appendix C.9

The Problem

- Using current technologies, memories can be either large or fast, but not both.
- The **size** or **capacity** of a memory refers to the number of bits it can store
- The **speed** of a memory typically refers to the amount of time it takes to access a stored entry: the delay between asking for the value stored at an address and receiving the result.

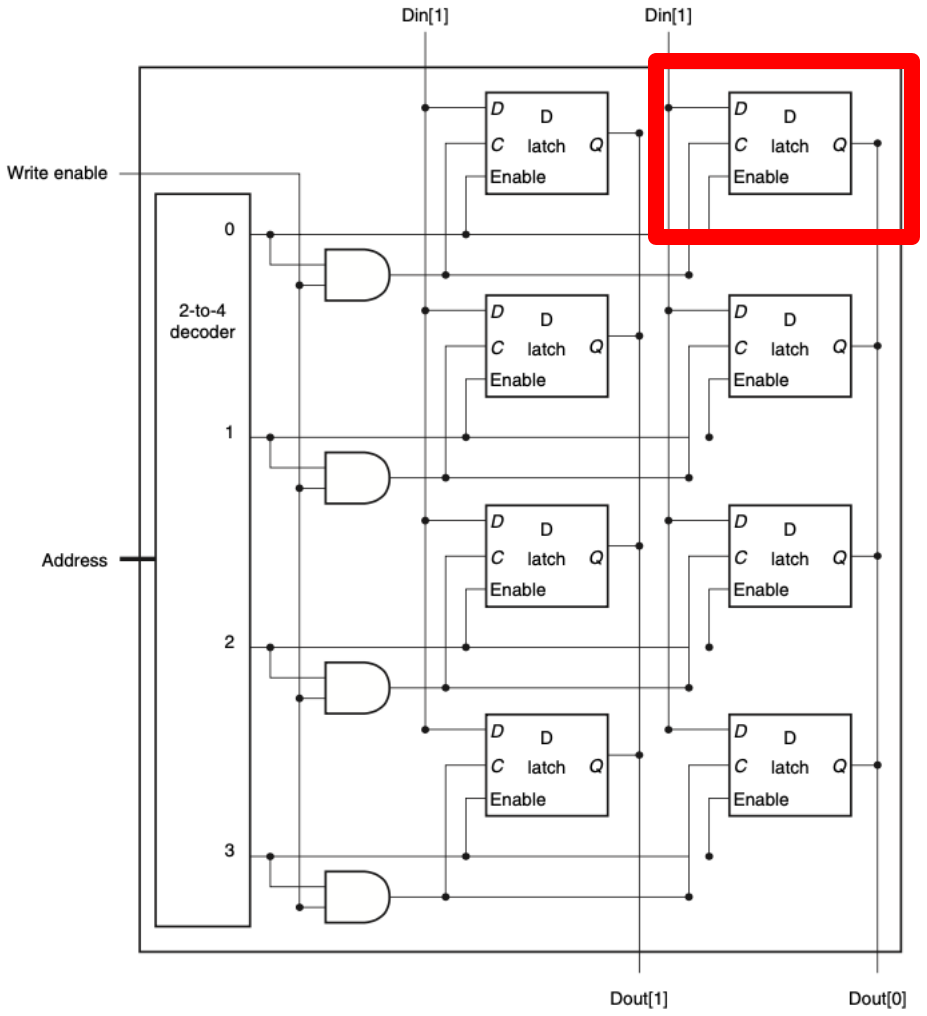
Memory Technologies

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 (~ sqrt(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 (>30ns internal access, ~50ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)
- Nonvolatile storage: Magnetic disk, NAND Flash, Phase-change memory, …

SRAM

- used in register file, caches



6 transistors per bit

FIGURE C.9.3 The basic structure of a 4 × 2 SRAM consists of a decoder that selects which pair of cells to activate. The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.

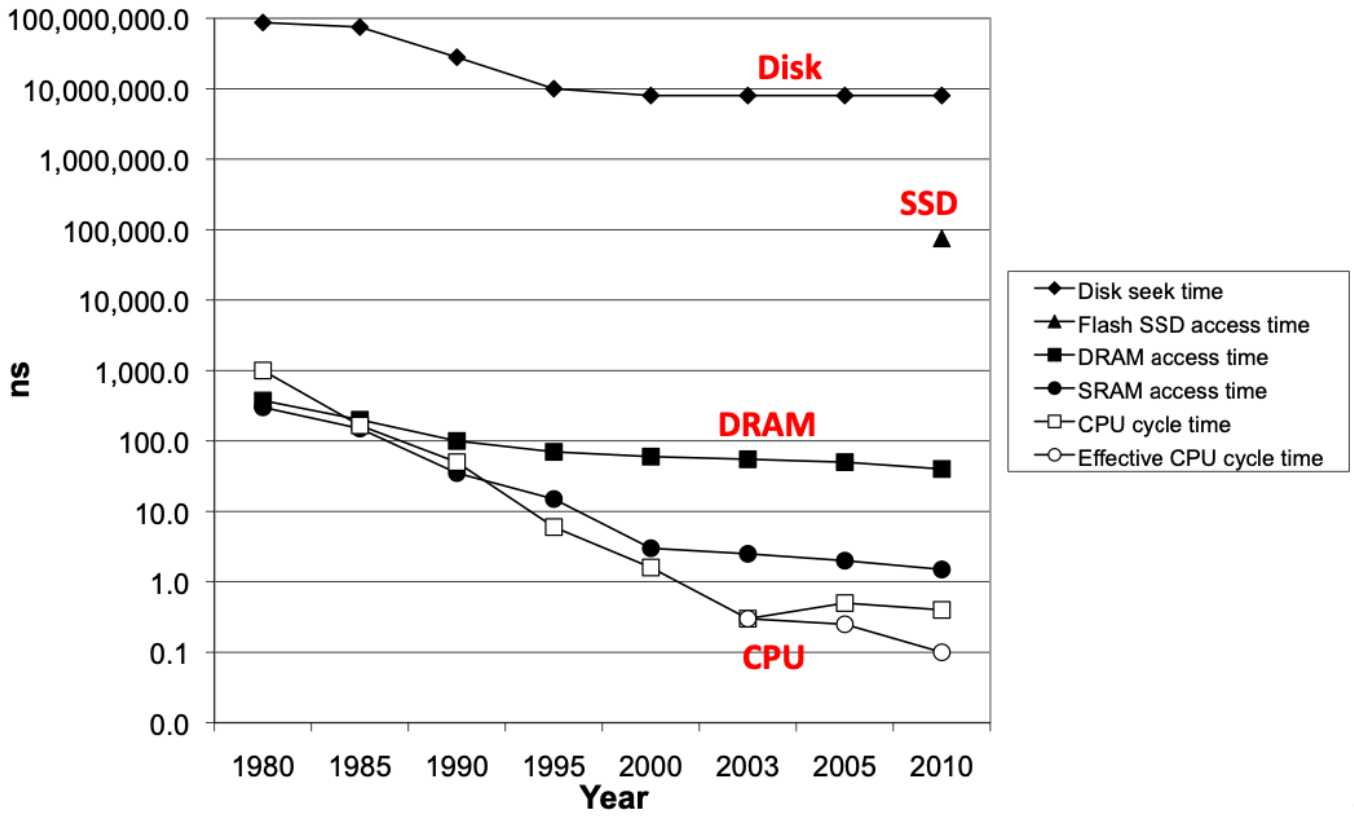
Memory Technology Trends

Carnegie Mellon

c/o O'Hallaron, Ganger, Kesden, CMU 15-213 / 18-213

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



47

Locality of Memory Technologies

- For many (all?) memory technologies, it may take a long time to get the first element out of the memory but you can fetch a lot of data in the vicinity very quickly
- It is usually a good idea to **buy in bulk** by fetching a lot of elements in the vicinity rather than just one.

The Memory Hierarchy

Big Picture 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

Known From the Beginning

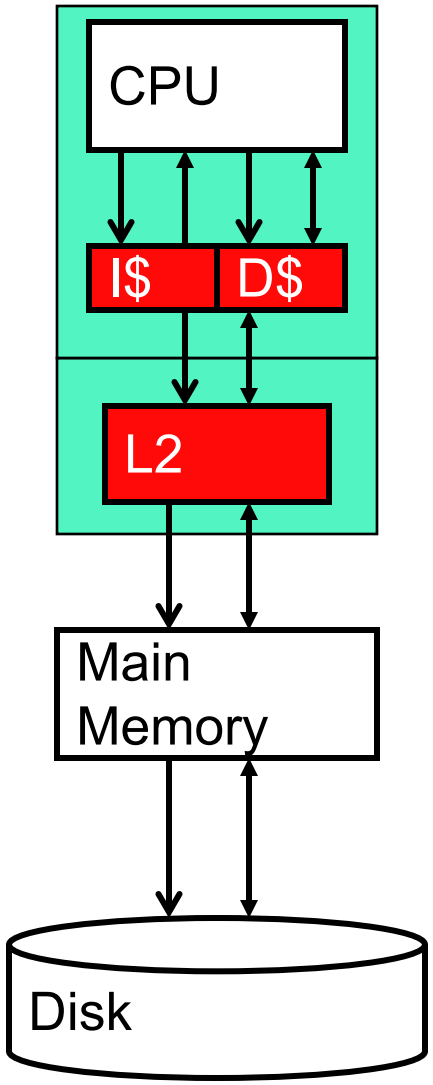
"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann

"Preliminary discussion of the logical design of an electronic computing instrument"

IAS memo **1946**

This Unit: Caches



- "Cache": hardware managed
  - Hardware automatically retrieves missing data
  - Built from fast SRAM, usually on-chip today
  - In contrast to off-chip, DRAM "main memory"
- Cache organization
  - Speed vs. Capacity
  - ABC
  - Miss classification
- Some example performance calculations

Key Observation: Locality

- **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: finite-sized hashtable of recently used data blocks
  - In hardware, transparent to software

Food Analogy

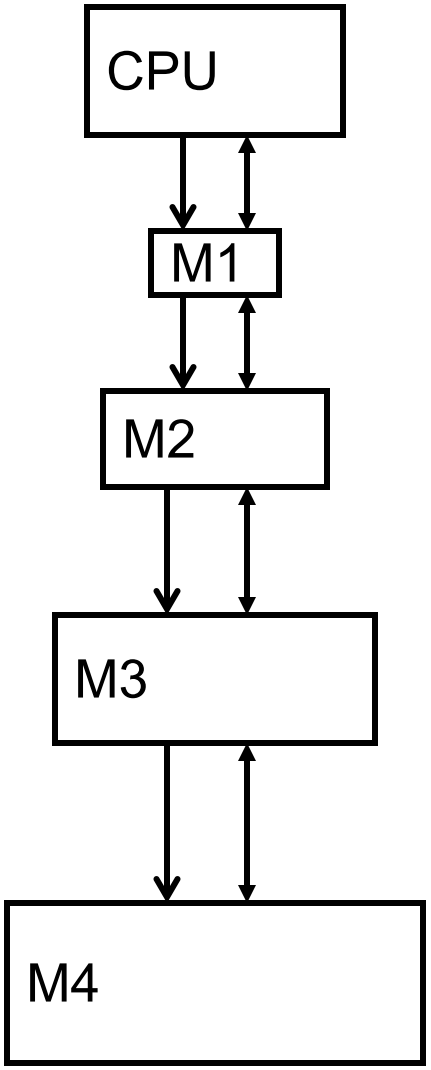
- How to get fast access to all of the food we might want?
- Fast access to food in your kitchen
  - but it has limited capacity
- The grocery store has much more food, but is slow
  - Far away (need to walk/drive)
  - Big (time to walk within the grocery store)
- How can you avoid these latencies?
  - My kitchen has a limited capacity
    - Keep recently used foods around (**temporal locality**)
    - Put related foods close together (**spatial locality**)
    - Guess what you'll need in the future (**prefetching**, later)

Spatial and Temporal Locality Example

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

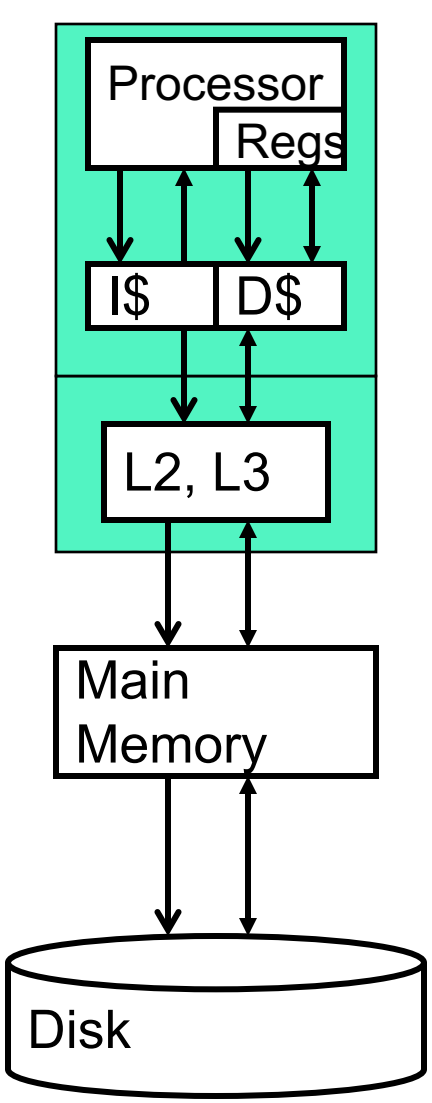
```
int sum = 0;
int X[1000];

for(int c = 0; c < 1000; c++){
  sum += X[c];
}
```

Exploiting Locality: Memory Hierarchy



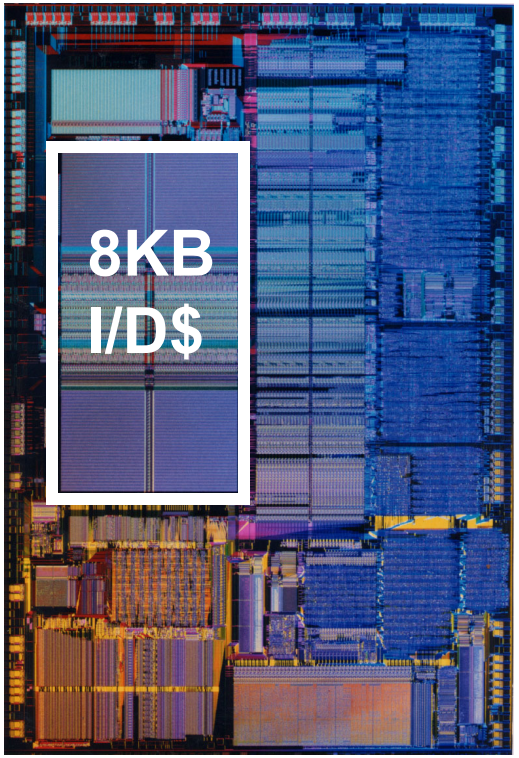
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by "buses"
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move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

Concrete Memory Hierarchy

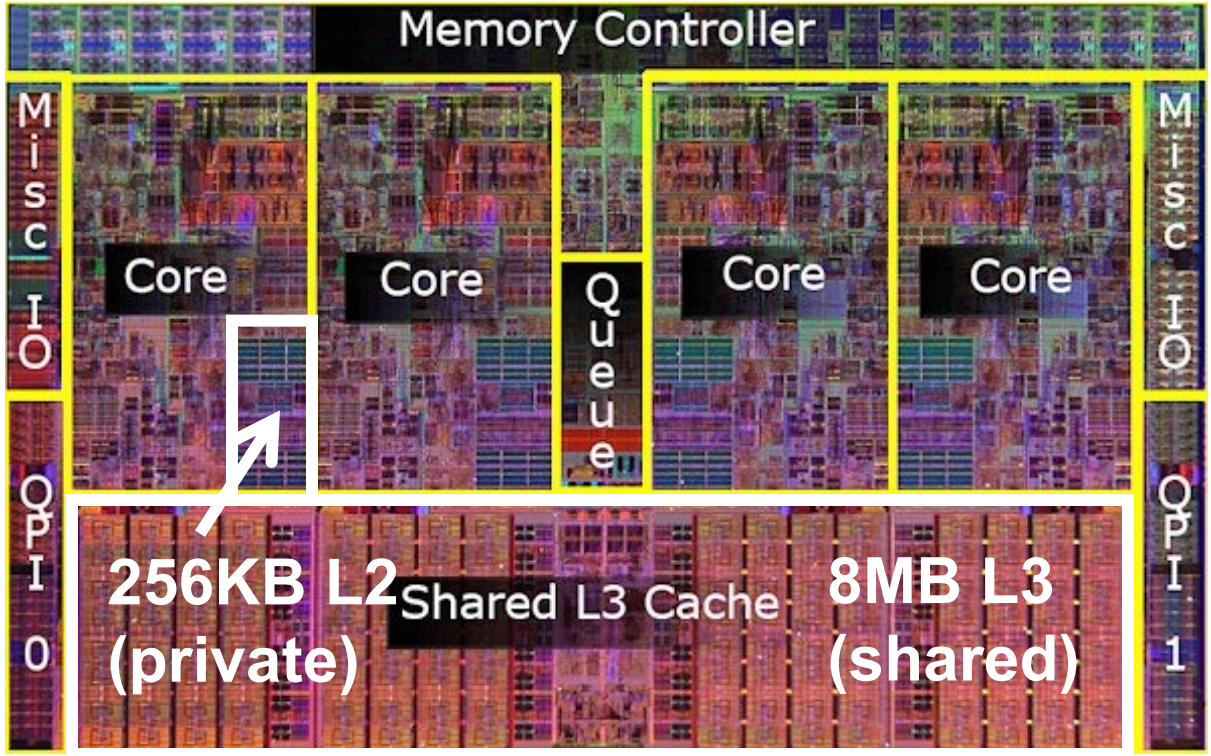


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

Evolution of Cache Hierarchies



Intel 486



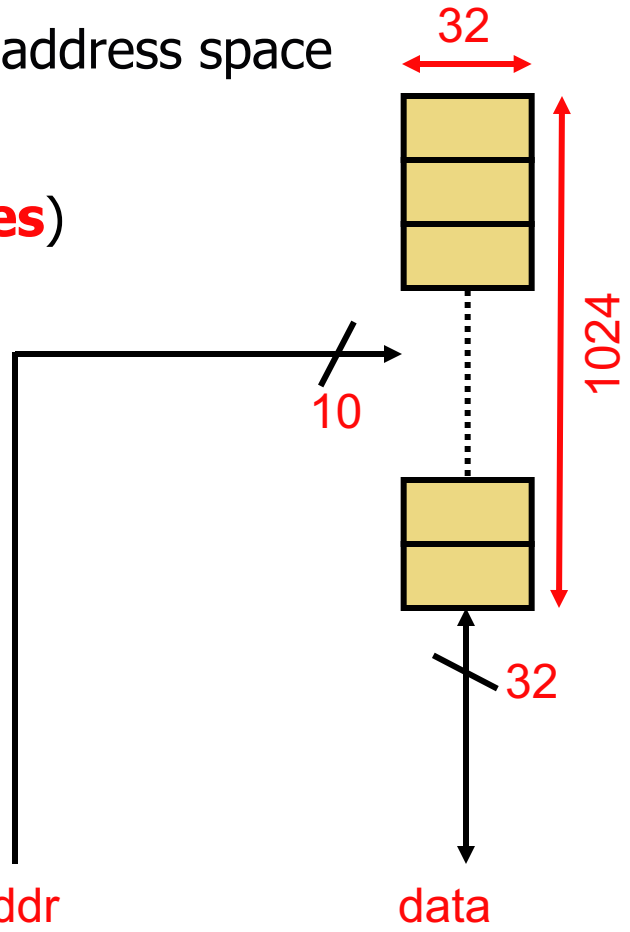
Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

Caches

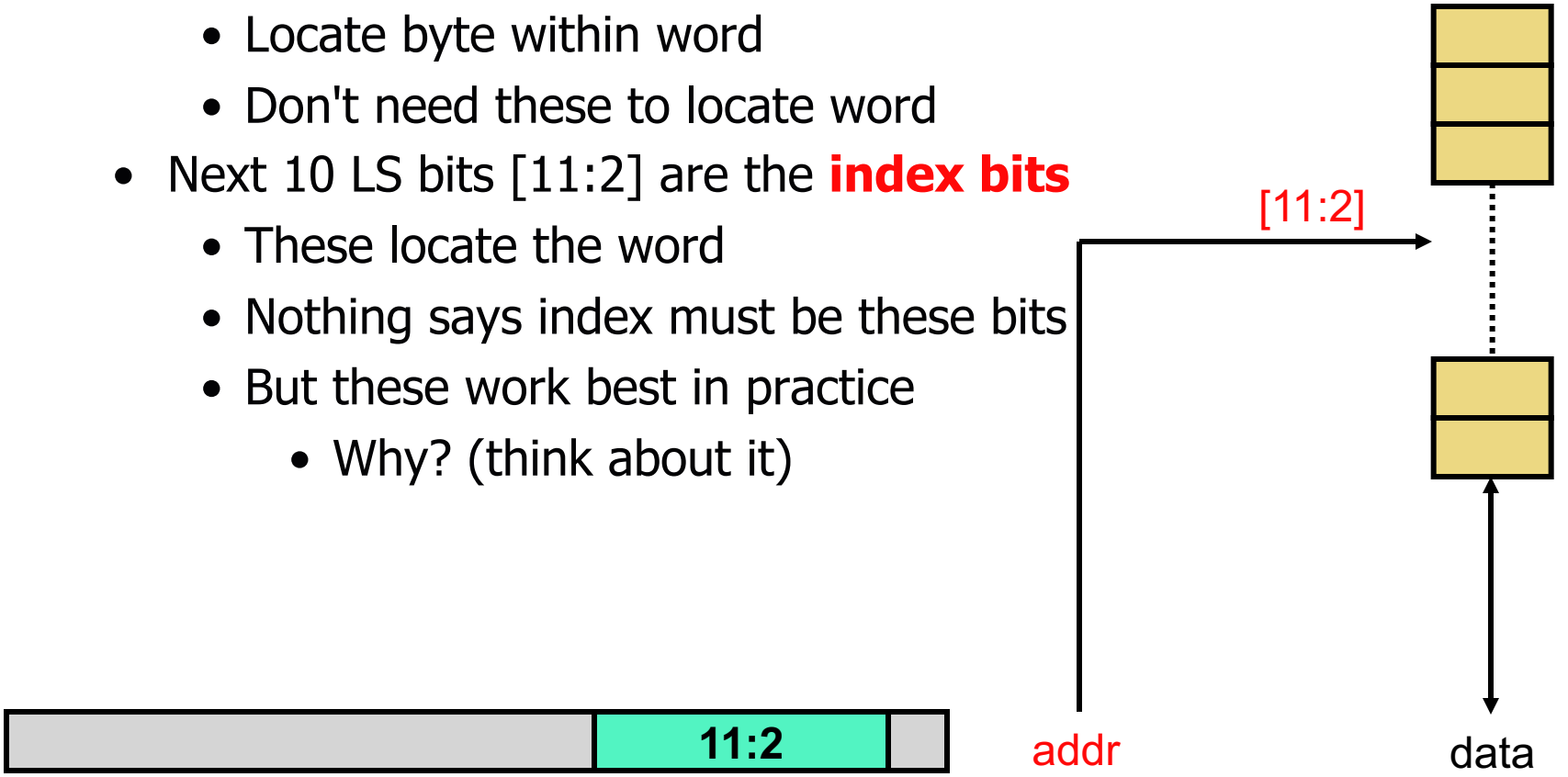
Logical Cache Organization

- **Cache is a hardware hashtable**
- The setup
  - 32-bit ISA $\rightarrow$ 4G words/addresses, 2^{32} B address space
- Logical cache organization
  - 4KB, organized as 1K 4B **blocks** (aka **lines**)
  - Each block can hold a 4-byte 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)

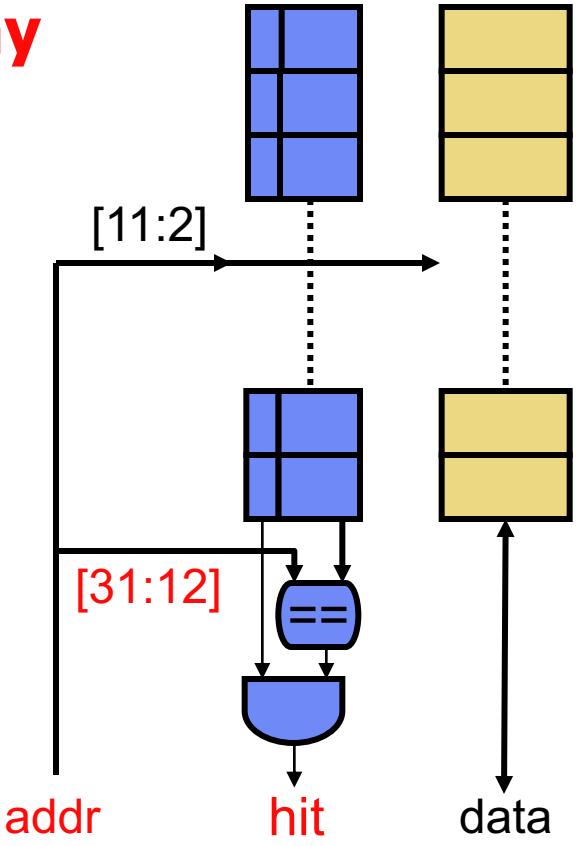


Is this the block you're looking for?

- Each cache row corresponds to 2^{20} blocks
  - 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:

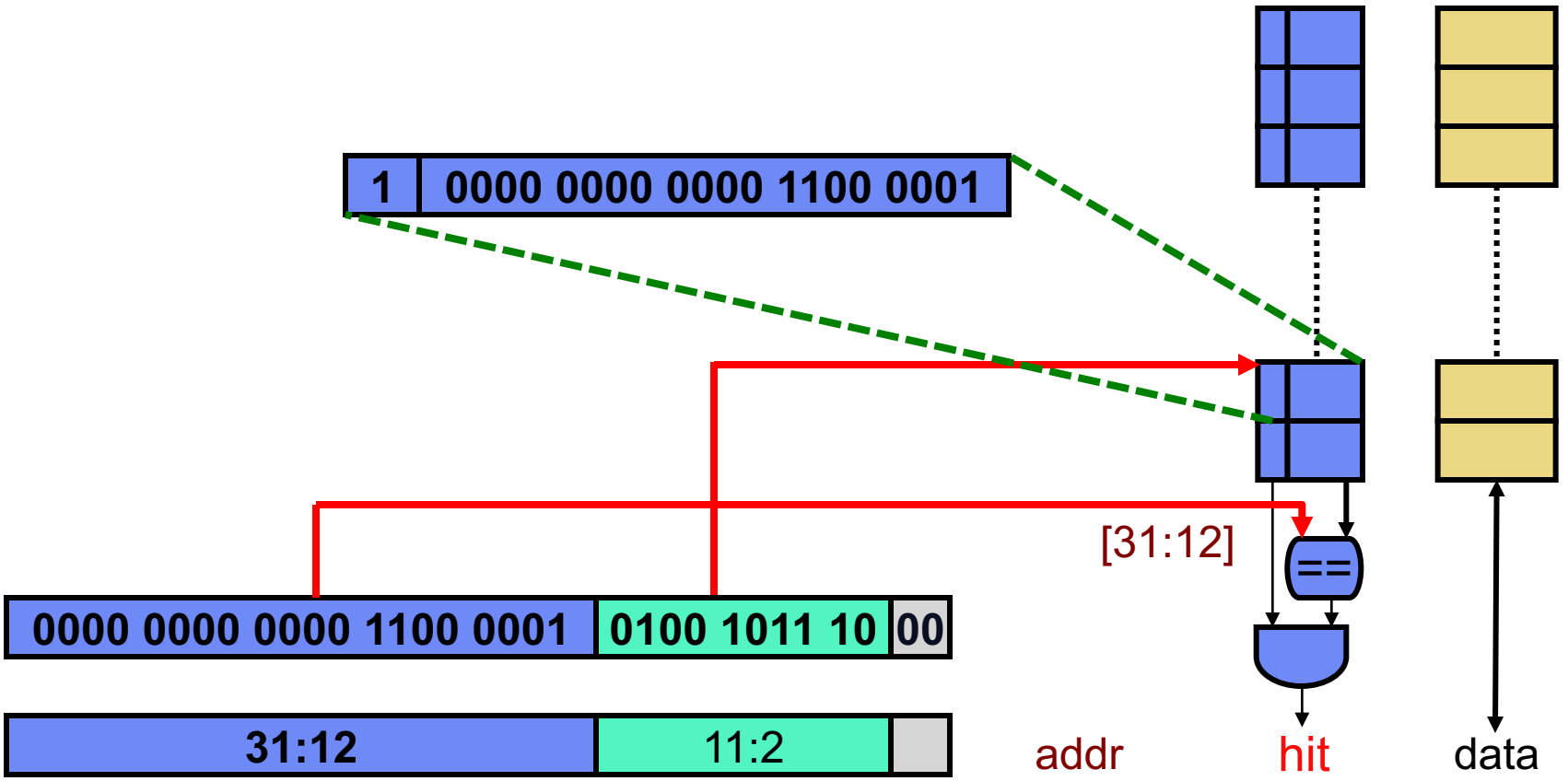
    then: Hit → data is good

    else: Miss → data is garbage, wait…



A Concrete Example

- Lookup address `x000C14B8`
  - Index = addr [11:2] = (addr >> 2) & x3FF = `x12E`
  - Tag = addr [31:12] = (addr >> 12) = `x000C1`



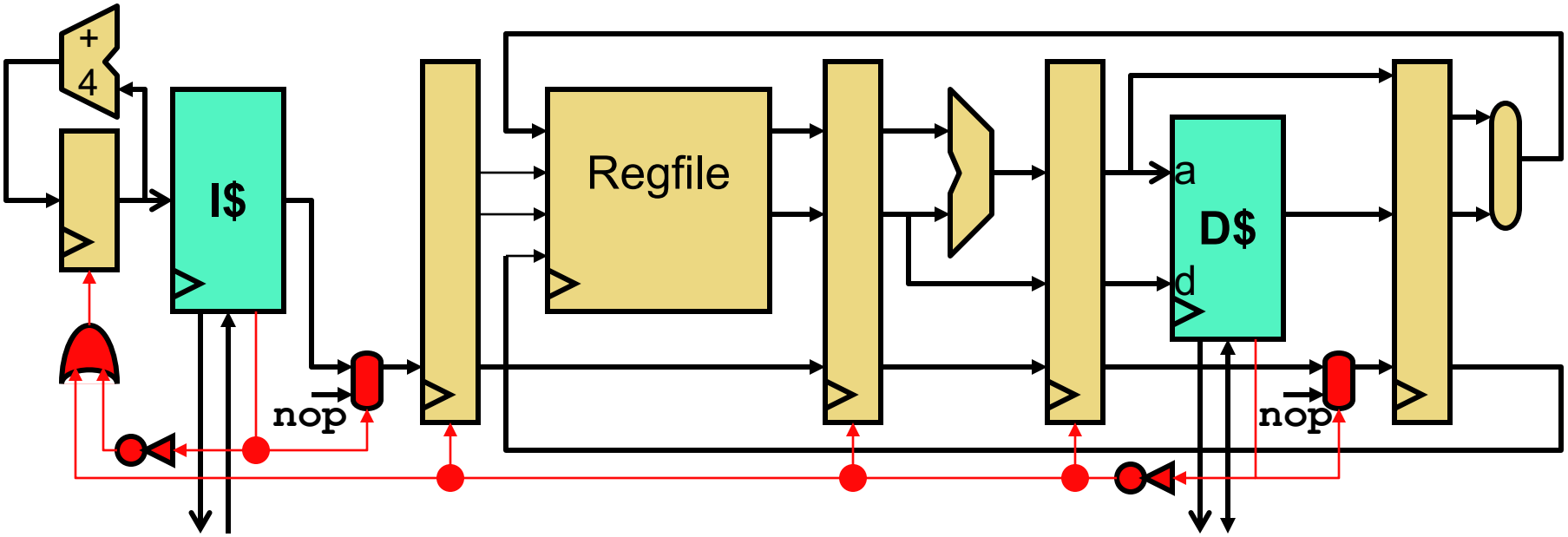
Handling a Cache Miss

- 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

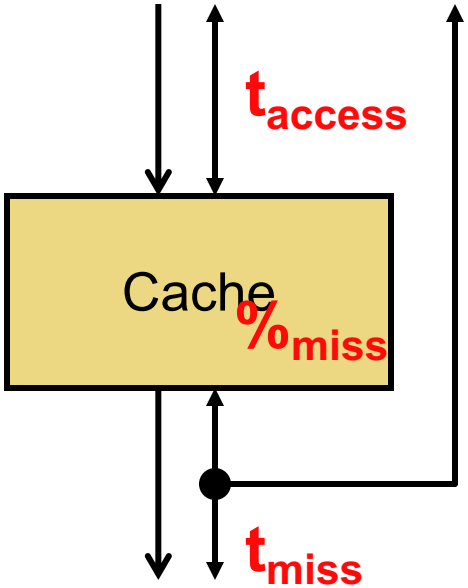
- Bringing a missing block into the cache is a **cache fill**

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 goes low → 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_{access}**: time to check cache. If hit, we're done.
  - **t_{miss}**: time to read data into cache

- Performance metric: average access time

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

Measuring Cache Performance

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

Cache operation: 1B block

- 8-bit addresses → 256B memory
  - Keeps diagrams simple

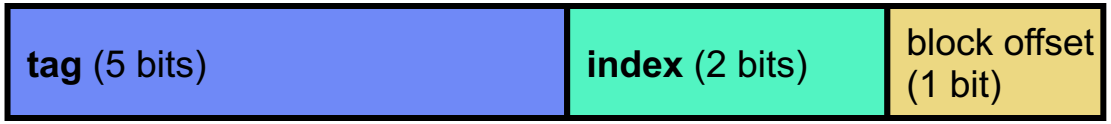
- 4B cache, 1B blocks
  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(1) = 0$
    - Index: next $\log_2$(number-of-sets) = $\log_2(4) = 2 \rightarrow$ 000000**00**
    - Tag: rest = 8 – 2 = 6 → **000000**00

Multi-Word Cache Blocks

- In most modern implementation we store more than one address (>1 byte) in each cache block.
- The number of bytes or words stored in each cache block is referred to as the **block size**.
- The entries in each block come from a contiguous set of addresses to exploit locality of reference, and to simplify indexing
- **Cache blocks** are also referred to as **cache lines**
  - Related to **cache frames** – a **frame** is the bucket, and the **block** is the data that goes in the bucket
  - blocks move around due to fills & evictions
    - frames are part of the cache structure and never move

Tag, Index, Block Offset

- Consider 8B cache with 2B blocks
  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(2) = 1 \rightarrow$ 0000000**0**
    - Index: middle $\log_2$(number-of-sets) = $\log_2(4) = 2 \rightarrow$ 00000**00**0
    - Tag: remaining high-order bits = $8 - 1 - 2 = 5 \rightarrow$ **00000**000

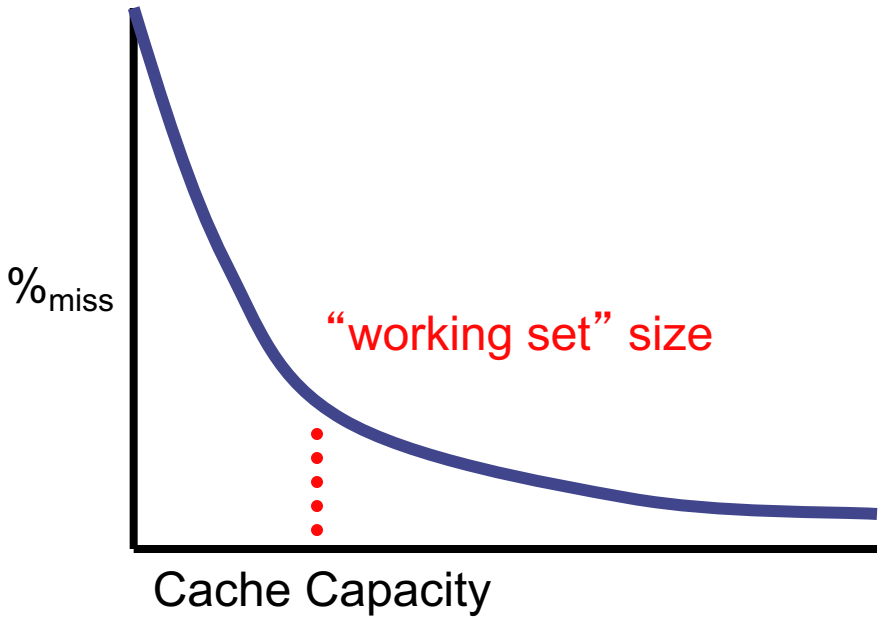


example

- 8B DM cache with 2B blocks

Capacity and Performance

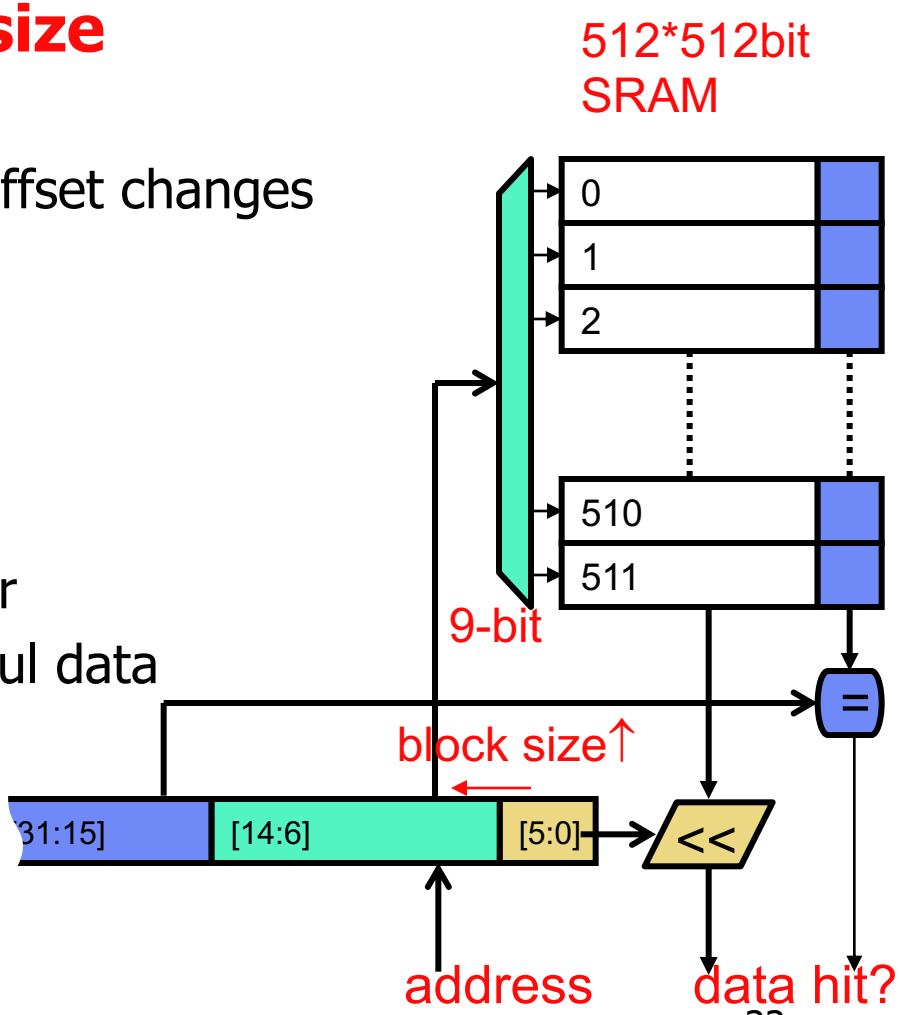
- Simplest way to reduce $\%_{miss}$: increase capacity
  - + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However t_{access} increases
    - Latency proportional to sqrt(capacity)



- Given capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size

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



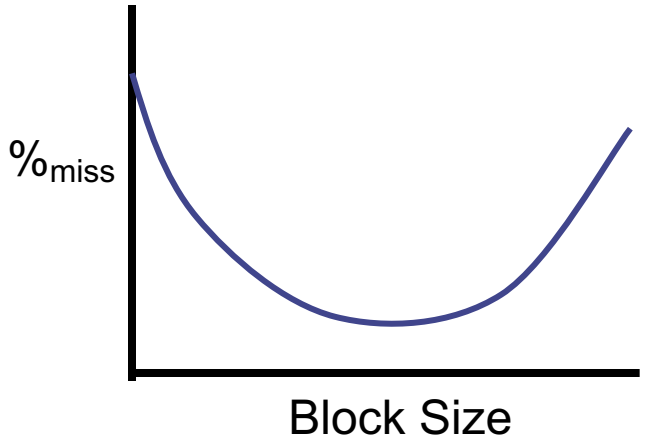
Block Size and Tag Overhead

- 4KB cache with 1024 4B blocks?
  - 4B blocks → 2-bit offset, 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = 63% overhead
- 4KB cache with 512 8B blocks
  - 8B blocks → 3-bit offset, 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - **20-bit tag** / **64-bit block** = **32% overhead**
  - Notice: tag size is same, but data size is twice as big
- A realistic example: 64KB cache with 64B blocks
  - 16-bit tag / 512-bit block = **~ 2% overhead**

- **Note: Tags are not optional**

Effect of Block Size on Miss Rate

- Two effects on miss rate
  - + **Spatial prefetching (good)**
    - For blocks with adjacent addresses
    - Turns miss/miss into miss/hit pairs
  - – **Interference (bad)**
    - For blocks with non-adjacent addresses (but in 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
  - Good block size is 32–256B
    - Program dependent



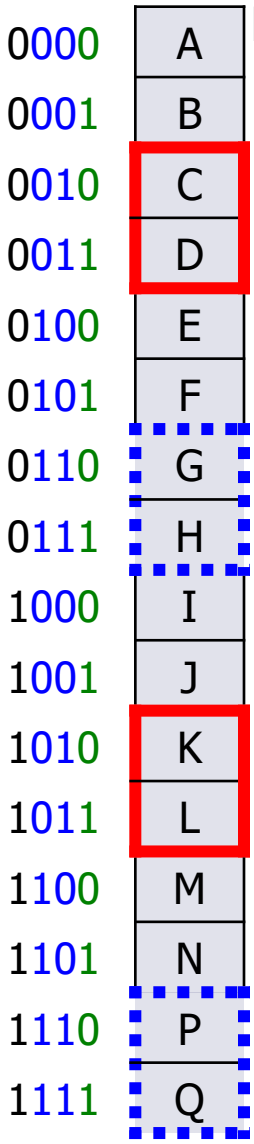
example

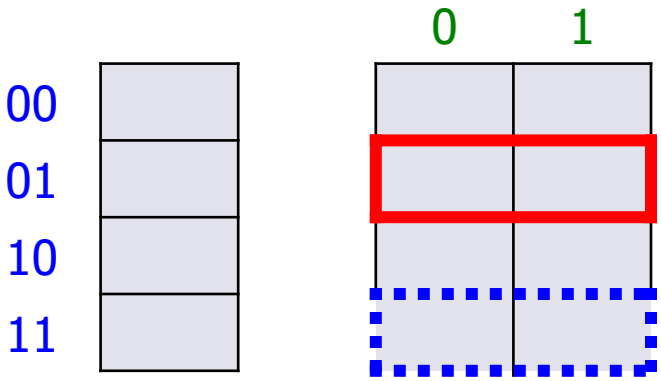
- 8B DM cache with 4B blocks

Block Size and Miss Penalty

- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses can't happen at the same time
  - Latencies can start to pile up
  - This is a bandwidth problem

Cache Conflicts



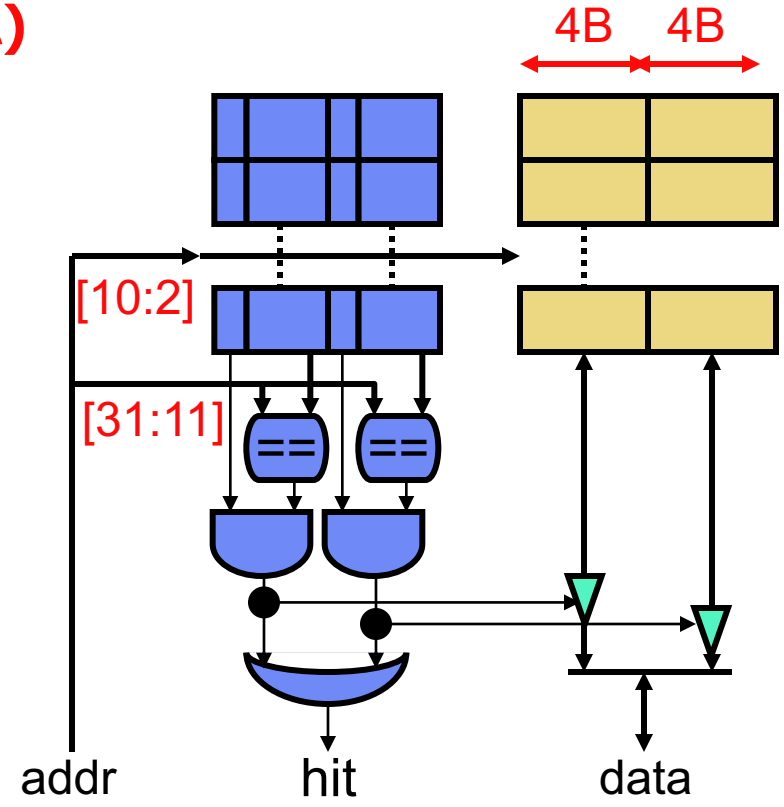


- Pairs like "0010" and "1010" **conflict**
  - **Same index!**
- Can such pairs to simultaneously reside in cache?
  - A: Yes, if we reorganize cache to do so

Associativity

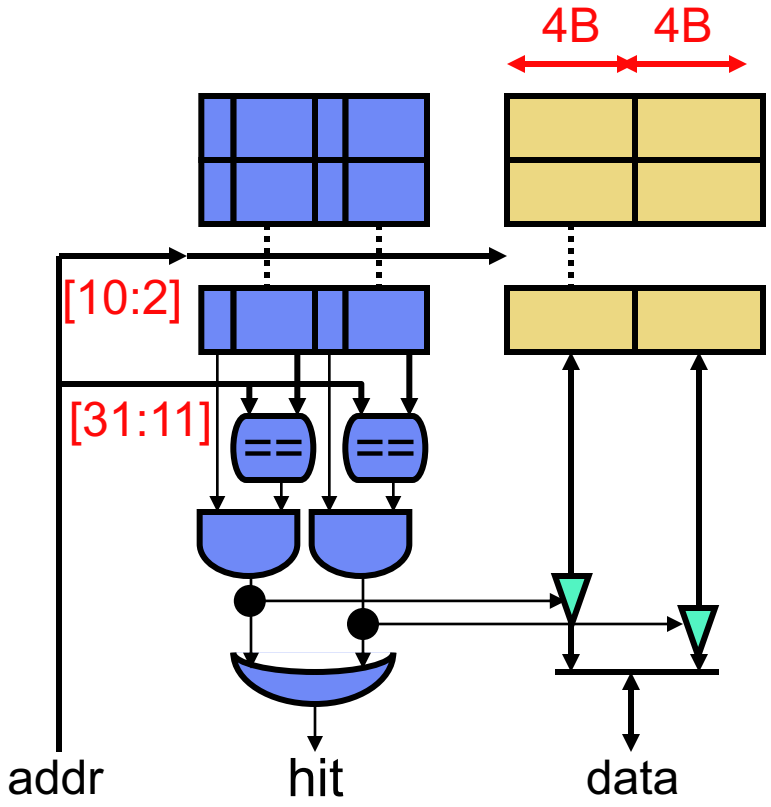
- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called ***sets***
  - Each frame in set called a ***way***
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

  + Reduces conflicts
  – Increases t_{access}:
    - 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)

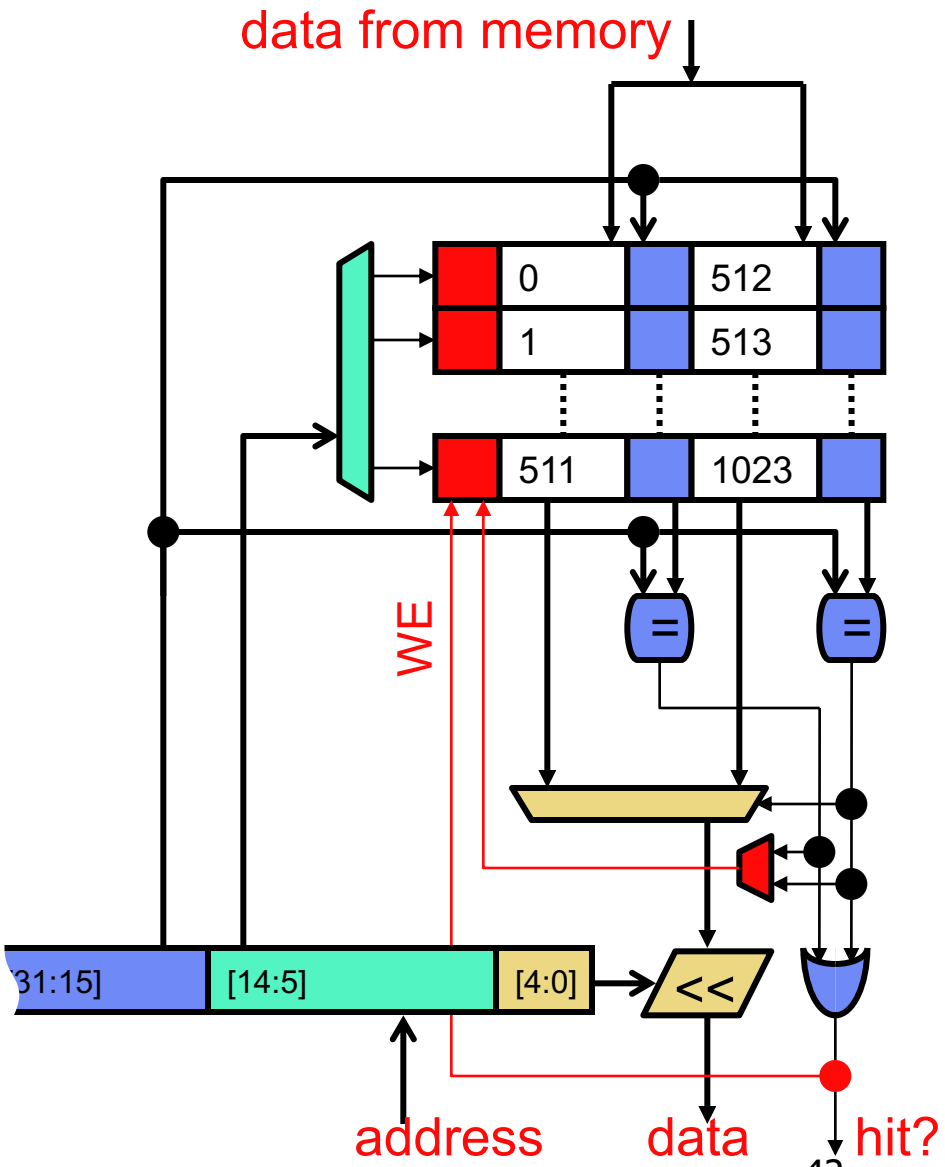


Replacement Policies

- 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
    - Is LRU for 2-way set-associative caches
  - **Belady's**: replace block that will be used furthest in future
    - Unachievable optimum

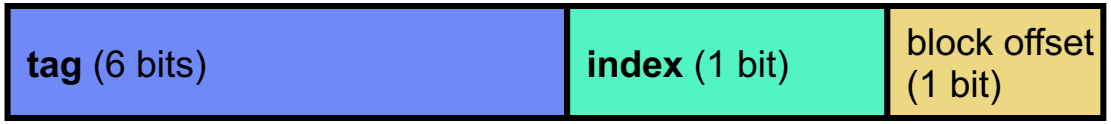
LRU and Miss Handling

- Add **LRU** field to each set
  - "Least recently used"
  - LRU data is encoded "way"
  - Hit? update MRU
- LRU bits updated on each access



2-way Set-Associative Cache Operation

- 8B cache, 2 ways, 2B blocks
  - Figure out number of sets: 2 ((capacity / ways) / block-size)
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(2) = 1 \rightarrow$ 0000000**0**
    - Index: next $\log_2$(number-of-sets) = $\log_2(2) = 1 \rightarrow$ 000000**0**0
    - Tag: rest = 8 – 1 – 1 = 6 $\rightarrow$ **000000**00

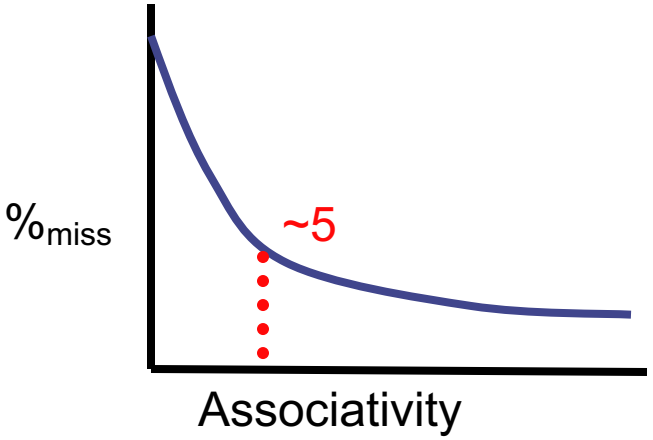


example

- http://comparchviz.com/cache.html

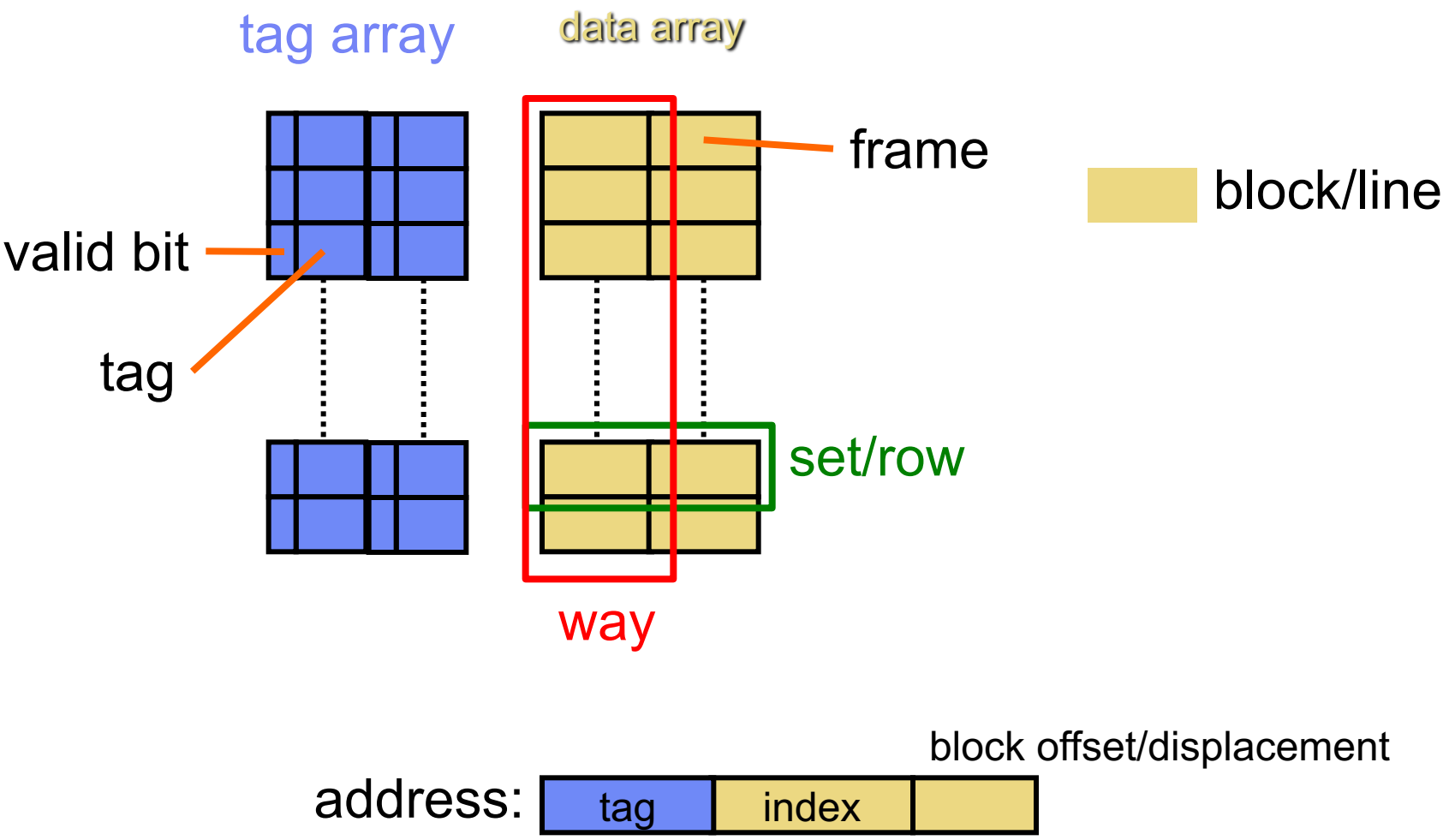
Associativity and Performance

- Higher associative caches
  - + Have better (lower) $\%_{miss}$
    - Diminishing returns
  - – However t_{access} 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

Cache Glossary



What About Stores? Handling Cache Writes

Write Issues

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache?

- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding write miss latency

Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is wrong (OK, just stall until good data arrives)

- Writes: read tag, write data in parallel? No! Why not?
  - Tag mis-match → clobbered data (oops!)
  - For associative caches, which way was written into?

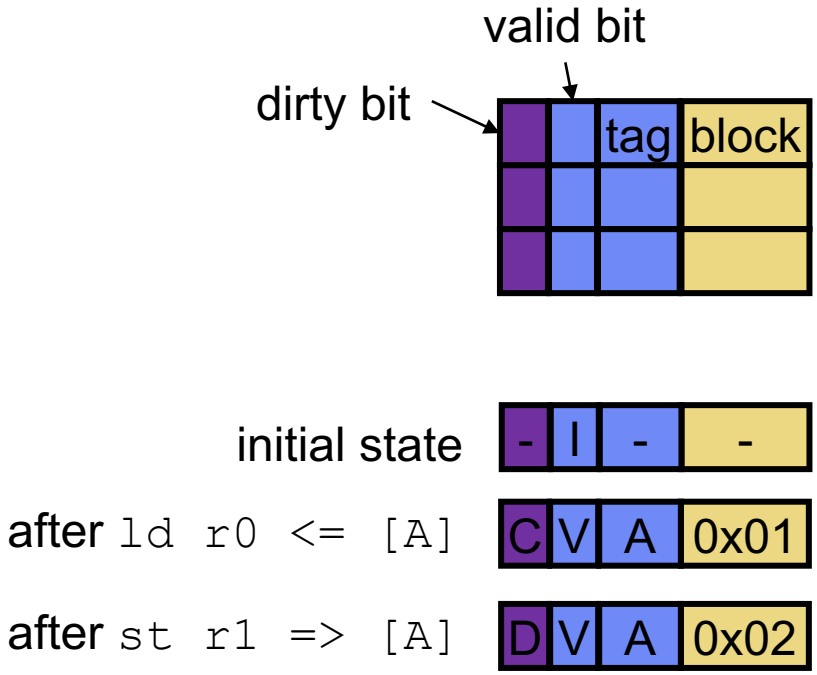
- Writes are a two step (multi-cycle) process
  - Step 1: match tag
  - Step 2: write to matching way
  - Bypass (with address check) to avoid load stalls
  - May introduce structural hazards

Write Propagation

- When to propagate new value to lower-level caches/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
  - Now we have multiple versions of the same block in various caches and in memory!
  - Requires additional "**dirty**" bit per block
    - Evict **clean** block: **no extra traffic**
      - there was only 1 version of the block
    - Evict **dirty** block: **extra "writeback" of block**
      - the dirty block is the most up-to-date version

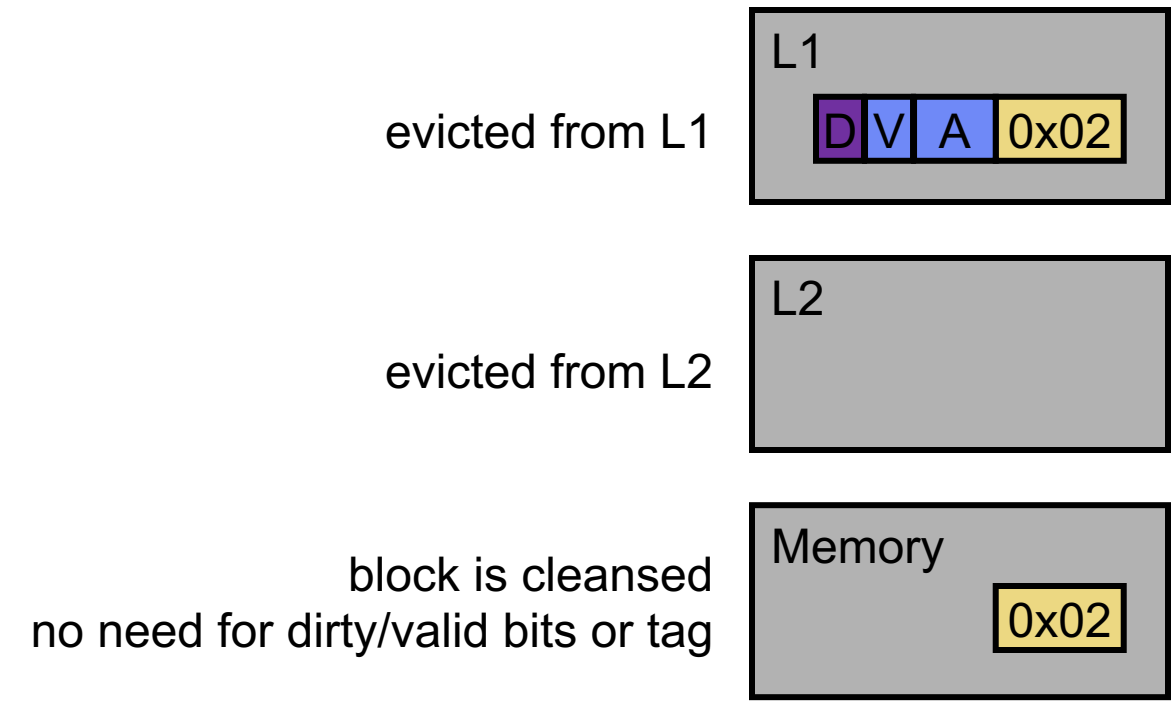
Write-back Cache Operation

- Each cache block has a **dirty bit** associated with it
  - state is either **clean** or **dirty**



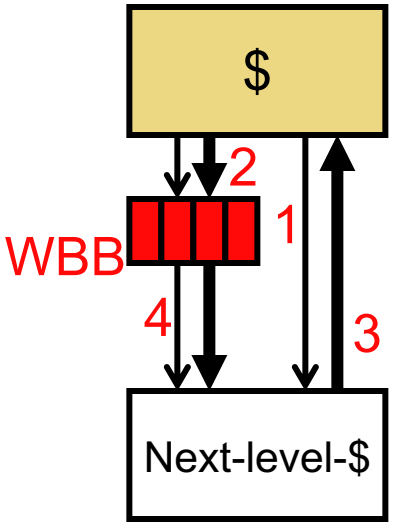
Write-backs across caches

- When a dirty block is evicted to a lower-level cache, it **remains** dirty
  - Writing a block back to memory **cleanses** it
  - There are never dirty blocks in memory, only in caches



Optimizing Writebacks

+ **Writeback-buffer (WBB)**:
  - Hide latency of writeback (keep them 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



Write Propagation Comparison

- **Write-through**
  - – Requires additional bus bandwidth
    - 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
    - Used in GPUs, as they have low write temporal locality

- **Write-back**
  - \+ Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used in most CPU designs

Write Miss Handling

- How is a write miss actually handled?

- **Write-allocate**: fill block from next level, then write it
  - \+ Decreases read misses (next read to block will hit)
  - – Requires additional bandwidth
  - Commonly used (especially with write-back caches)

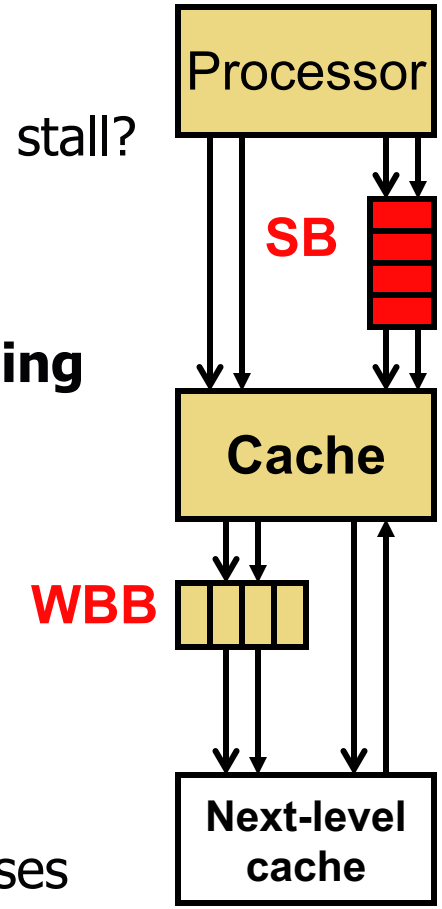
- **Write-non-allocate**: just write to next level, no allocate
  - – Potentially more read misses
  - \+ Uses less bandwidth
  - Use with write-through

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
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems for multicore (later)

- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Improving Effectiveness of Memory Hierarchy

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 caused 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 caused because cache associativity is too low
    - Identify? **All other misses**
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)

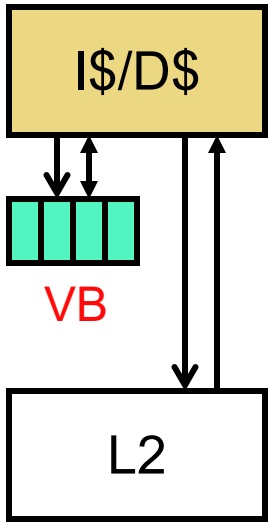
Miss Rate: ABC

- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don't have conflict misses, increasing associativity won't help

- More **Associativity** (assuming fixed capacity)
  - \+ Decreases conflict misses
  - – Increases t_{access}
- Larger **Block size** (assuming fixed capacity)
  - – Increases conflict/capacity misses (fewer frames)
  - \+ Decreases compulsory misses (spatial locality)
  - No significant effect on t_{access}
- More **Capacity**
  - \+ Decreases capacity misses
  - – Increases t_{access}

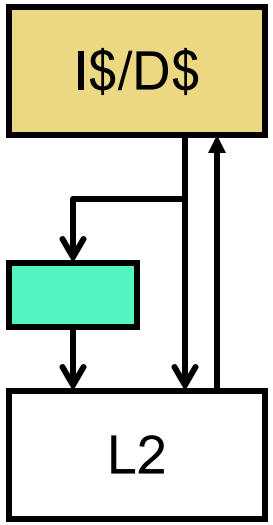
Reducing Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small (e.g., 8 entries) so very fast
  - Blocks kicked out of I$/D$ 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



Prefetching

- Bring data into cache proactively/**speculatively**
  - If successful, reduces number of caches misses
- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware
- Simple hardware prefetching: **next block prefetching**
  - Miss on address **X** $\rightarrow$ 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
  - Just a hint
- Inserted by programmer or compiler
- Example

```
int tree_add(tree_t* t) {
  if (t == NULL) return 0;
  __builtin_prefetch(t->left);
  __builtin_prefetch(t->right);
  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
  - – Compiler must know that restructuring preserves semantics

- **Loop interchange**: spatial locality
  - Example: row-major matrix: `X[i][j]` 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
```

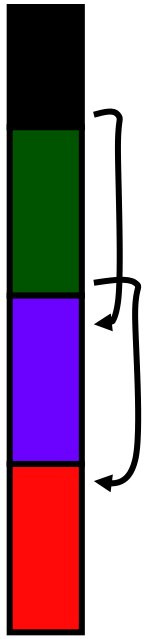
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

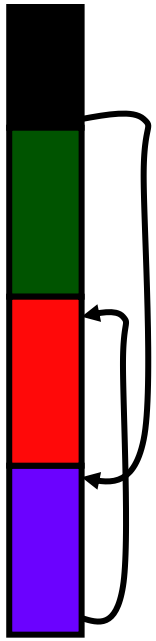
```
for (i=0; i<NUM_ELEMS; i+=CACHE_SIZE)
    for (k=0; k<NUM_ITERATIONS; k++)
        for (j=0; j<CACHE_SIZE; j++)
            X[i+j] = f(X[i+j]);
```

  – Assumes you know **CACHE_SIZE**, do you?

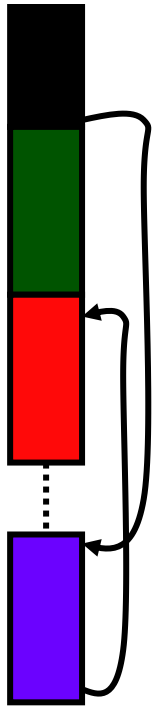
Software Restructuring: Code

- Compiler can lay out code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, **code2** case never happens (say, error condition)





+ Better locality
+ Fewer taken branches



+ Better locality for code after **code3**
+ Fewer taken branches

Cache Hierarchies

Designing a Cache Hierarchy

- For any memory component: t_{access} vs. $\%_{miss}$ tradeoff

- Upper components (I$, D$) emphasize low t_{access}
  - Frequent access $\rightarrow$ t_{access} important
  - t_{miss} is not bad $\rightarrow$ $\%_{miss}$ less important
  - Lower capacity and lower associativity (to reduce t_{access})
  - Small-medium block-size (to reduce conflicts)

- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access $\rightarrow$ t_{access} less important
  - t_{miss} is bad $\rightarrow$ $\%_{miss}$ important
  - High capacity, associativity, and block size (to reduce $\%_{miss}$)

Memory Hierarchy Parameters

Parameter	I$/D$	L2	L3	Main Memory
t_{access}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	2-8	4–16	4-16	NA

- Some other design parameters
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing

Split vs. Unified Caches

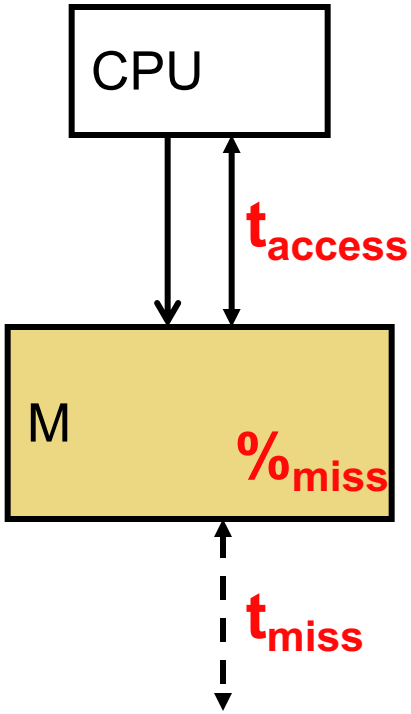
- **Split I$/D$**: insns and data in different caches
  - To minimize structural hazards and t_{access}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ and D$ separately
    - Not writes for I$, smaller reads for D$
  - Why is 486 I/D$ unified?

- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$
  + 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
  - 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

Memory Performance Equation



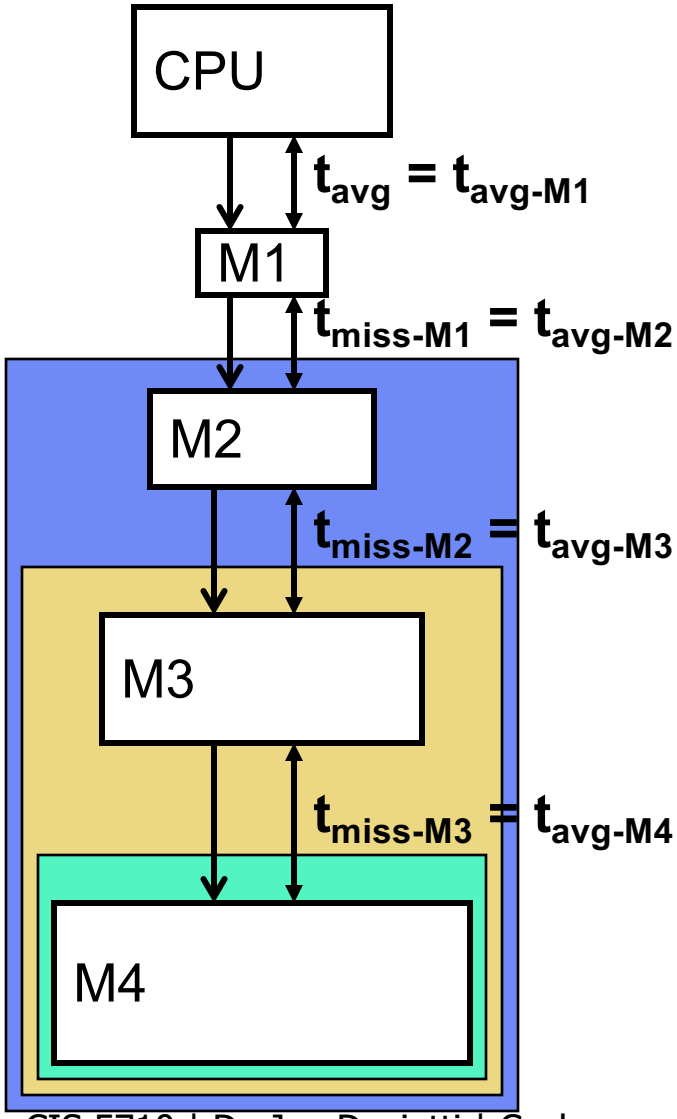
- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M

  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{access}: time to read data from (write data to) M
  - t_{miss}: time to read data into M

- Performance metric
  - t_{avg}: average access time

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

Hierarchy Performance



t_{avg}

t_{avg-M1}

$t_{acc-M1} + (\%_{miss-M1} * t_{miss-M1})$

$t_{acc-M1} + (\%_{miss-M1} * t_{avg-M2})$

$t_{acc-M1} + (\%_{miss-M1} * (t_{acc-M2} + (\%_{miss-M2} * t_{miss-M2})))$

$t_{acc-M1} + (\%_{miss-M1} * (t_{acc-M2} + (\%_{miss-M2} * t_{avg-M3})))$

…

Performance Calculation I

- In a pipelined processor, I$/D$ t_{access} 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 new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{memory} * \%_{missD\$} * t_{missD\$}$ = 0.30*0.10*10 cycles = 0.3 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 1/3rd 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 rate!)
    - L1 acts as a filter for L2 accesses

Multilevel Performance Calculation II

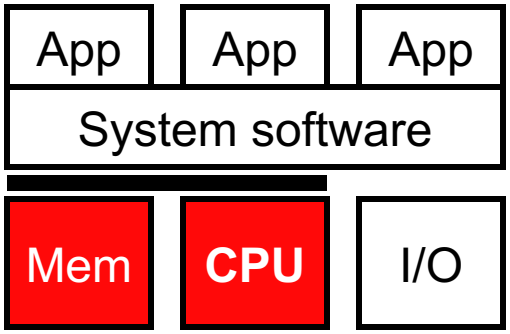
- Parameters
  - 30% of instructions are memory operations
  - L1: t_{access} = 1 cycles (included in CPI of 1), $\%_{miss}$ = 5% of accesses
  - L2: t_{access} = 10 cycles, $\%_{miss}$ = 20% of L2 accesses
  - Main memory: t_{access} = 50 cycles
- Calculate CPI
  - CPI = 1 + 30% * 5% * $t_{missD\$}$
  - $t_{missD\$} = t_{avgL2} = t_{accL2}+(\%_{missL2}*t_{accMem}) = 10 + (20\%*50) = 20$ cycles
  - Thus, 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
  - CPI = 1 + (1.5% * 10) + (0.3% * 50) = 1 + 0.15 + 0.15 = 1.3 CPI

Foreshadow: Main Memory As A Cache

Parameter	I$/D$	L2	L3	Main Memory
t_{acc}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8–64KB	128KB–2MB	1–9MB	64MB–64GB
Block size	16–32B	32–256B	256B	4KB+
Associativity	1–4	4–16	16	full
Replacement	LRU	LRU	LRU	"working set"
Prefetching?	Maybe	Probably	Probably	Either

- How would you internally organize main memory
  - t_{miss} is outrageously long, reduce $\%_{miss}$ at all costs
  - Full associativity: isn't that difficult to implement?
    - Yes … in hardware, but main memory is "software-managed"

Summary



- **Average access time** of a memory component
  - $t_{avg} = t_{access} + \%_{miss} * t_{miss}$
  - Hard to get low t_{access} and $\%_{miss}$ in one structure → build a hierarchy instead
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**associativity, block size, capacity**)
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: prefetching
  - t_{miss}: victim buffer, critical-word-first
- **Write issues**
  - Write-back vs. write-through, write-allocate vs. write-no-allocate