Example: CPI Calculation with Cache Misses

- Base CPI = 1
- Instruction mix: 30% loads/stores
- L1 I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
- L1 D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

} Considering cache misses, CPI = ?

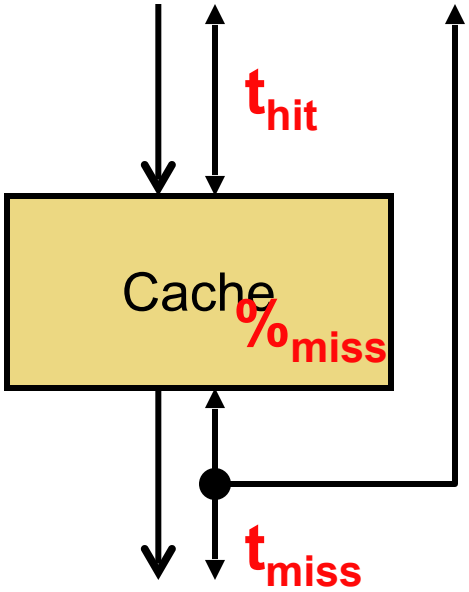
Assume 100 instructions in total → how many cycles to run?

- Without stalls, 100 instructions will take 100 cycles (base CPI = 1)
- All 100 instructions will need to access I$
  - 100 * 2% = 2 instructions miss in L1 I$ → each stall for 10 cycles
- 30 instructions are `lw` or `sw` that need to access D$
  - 30 * 10% = 3 instructions miss in L1 D$ → each stall for 10 cycles
- Total cycles stalled:
  - I$ stalls: 2 * 10 = 20 cycles
  - D$ stalls: 3 * 10 = 30 cycles

} Stall for 50 cycles in total

- Total cycles to execute the 100 instructions: 100 + 50 = 150 cycles
- CPI = 150 / 100 = 1.5

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
  - **$\%_{miss}$** (i.e., 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

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

AMAT = t_{avg} of memory access

Example: Use the equation to calculate CPI

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

- Simple pipeline with base CPI of 1
- Instruction mix: 30% loads/stores
- I\$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
- D\$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- What is CPI?
  - $CPI = CPI_{base} + (\%_{cache\_access}\%_{miss} * CPI_{miss})$
  - t_{hit} is already included in CPI_{base}

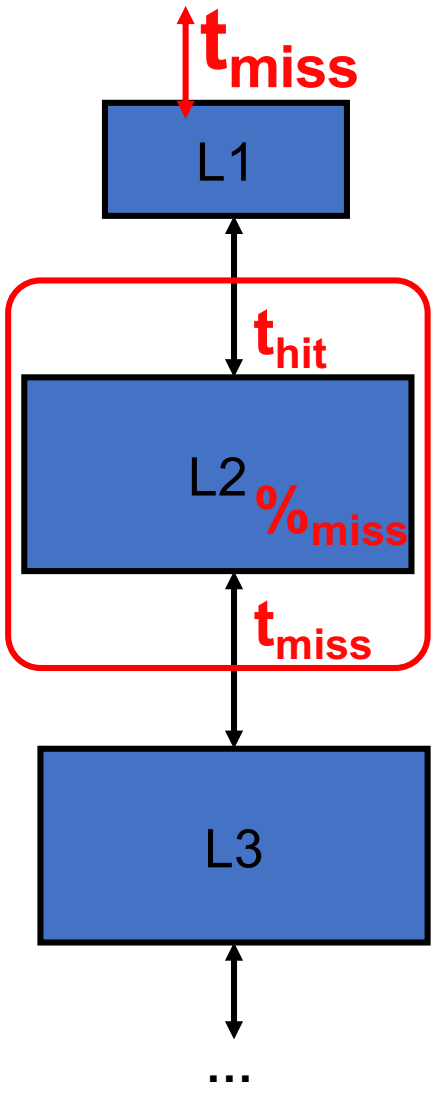
Example: Use the equation to calculate CPI

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

- Simple pipeline with base CPI of 1
- Instruction mix: 30% loads/stores
- I\$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
- D\$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- What is CPI?
  - $CPI = CPI_{base} + (\%_{cache\_access}\%_{miss} * CPI_{miss})$
  - CPI = 1 + (100%*2%*10 cycles/insn) + (30%*10%*10 cycles/insn)
    = 1 + 0.2 + 0.3
    = 1.5

How about multiple levels of caches



- For a cache
  - $\%_{miss}$ (i.e., miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) cache
  - t_{miss}: time to read data into cache

  *note: t can be *ns* or *cycles*
- Average time to access a level of cache

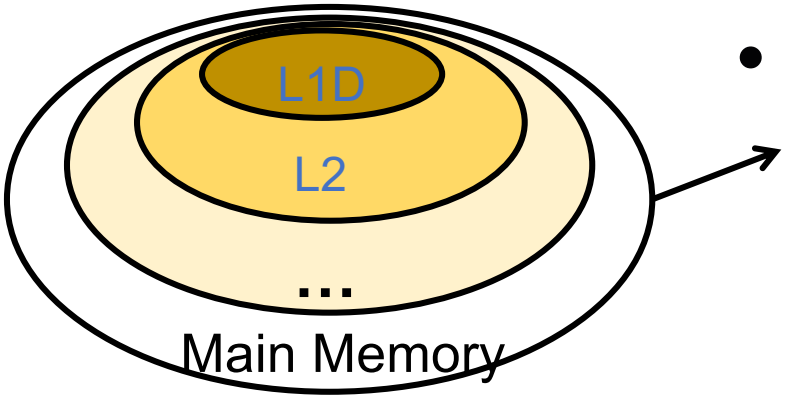
  $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
- t_{miss} = t_{avg} of next level →
- $t_{miss}(L1) = t_{hit}(L2) + \%_{miss}(L2)*t_{miss}(L2)$
-
- $= t_{hit}(L2) + \%_{miss}(L2)* (t_{hit}(L3) + \%_{miss}(L3)*t_{miss}(L3)$
- $= \ldots$

Multilevel Performance Calculation

- Parameters
  - 30% of instructions are memory operations
  - L1: t_{hit} = 1 cycles (included in CPI of 1), $\%_{miss}$ = 5% of accesses
  - L2: t_{hit} = 10 cycles, $\%_{miss}$ = 20% **of L2 accesses**
  - Main memory: t_{hit} = 50 cycles
- Calculate CPI
  - CPI = 1 + 30% * 5% * $t_{missD\$}$
  - $t_{missD\$} = t_{avgL2} = t_{hitL2}+(\%_{missL2}*t_{hitMem}) = 10 + (20\%*50) = 20$ 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

Types of cache

Inclusive vs. exclusive caches



- Inclusive cache
  - A higher-level of cache stores a subset of lower-level caches

- Exclusive cache (e.g., AMD Athlon)
  - Data is guaranteed to be in at most one of the L1 and L2 caches, never in both
- Non-inclusive cache (e.g., Intel Pentium II, III, and 4)
  - Intermediate policy
  - Do not require that data in the L1 cache also reside in the L2 cache, although it may often do so

Cheng et al., "LAP: Loop-Block Aware Inclusion Properties for Energy-Efficient Asymmetric Last Level Caches", ISCA 2016.

Directly vs. Associative Mapped Caching

- Direct mapped caching allows any given main memory block to be mapped into exactly one unique cache location.

- Set-associative mapped cache allows any given main memory block to be mapped into two or more cache locations.

- Fully-associative mapped caching allows any given main memory block to be mapped into any cache location.

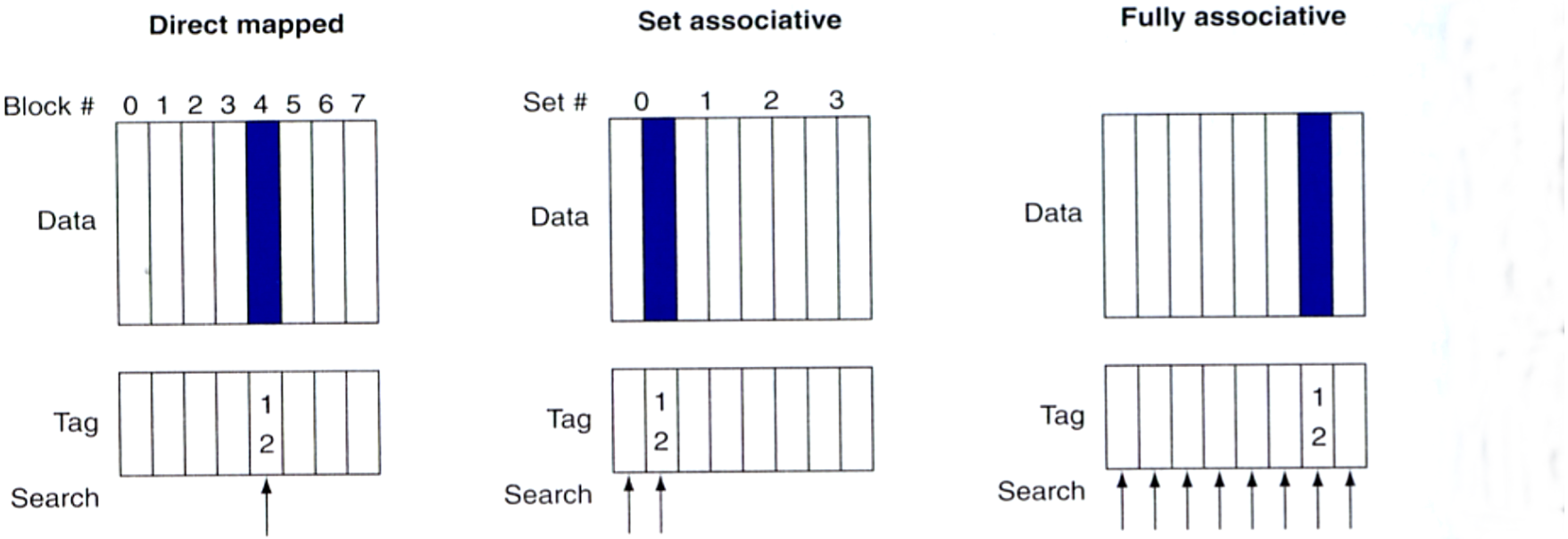


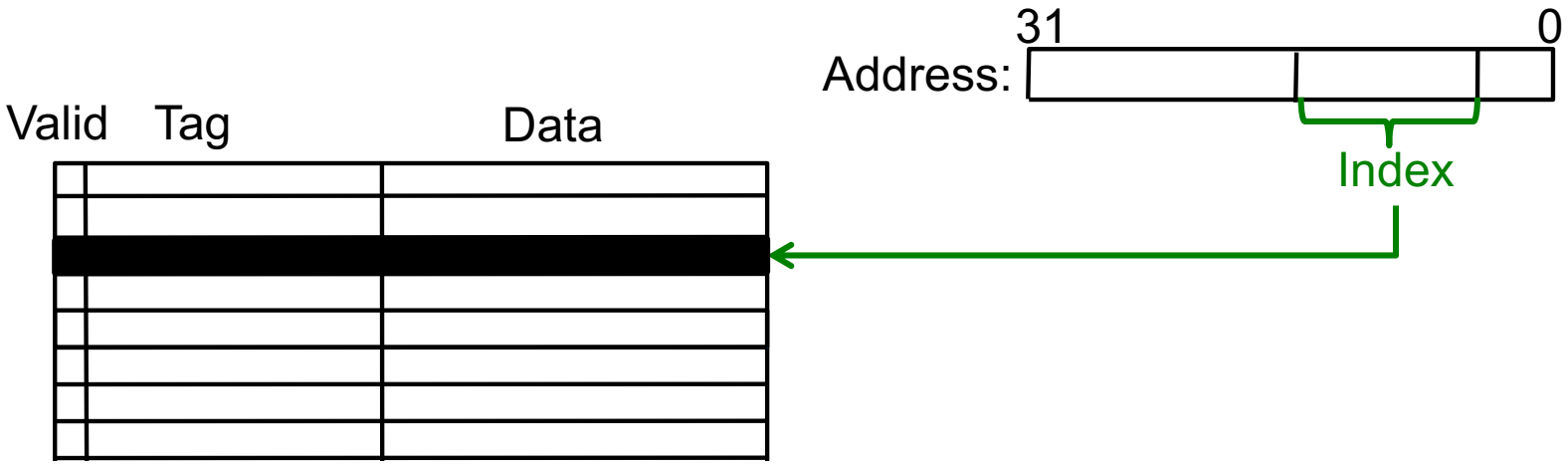
FIGURE 7.13 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped,

Directly vs. Associative Mapped Caching

- Direct mapped caching allows any given main memory block to be mapped into exactly one unique cache location.

- Set-associative mapped cache allows any given main memory block to be mapped into two or more cache locations.

- Fully-associative mapped caching allows any given main memory block to be mapped into any cache location.

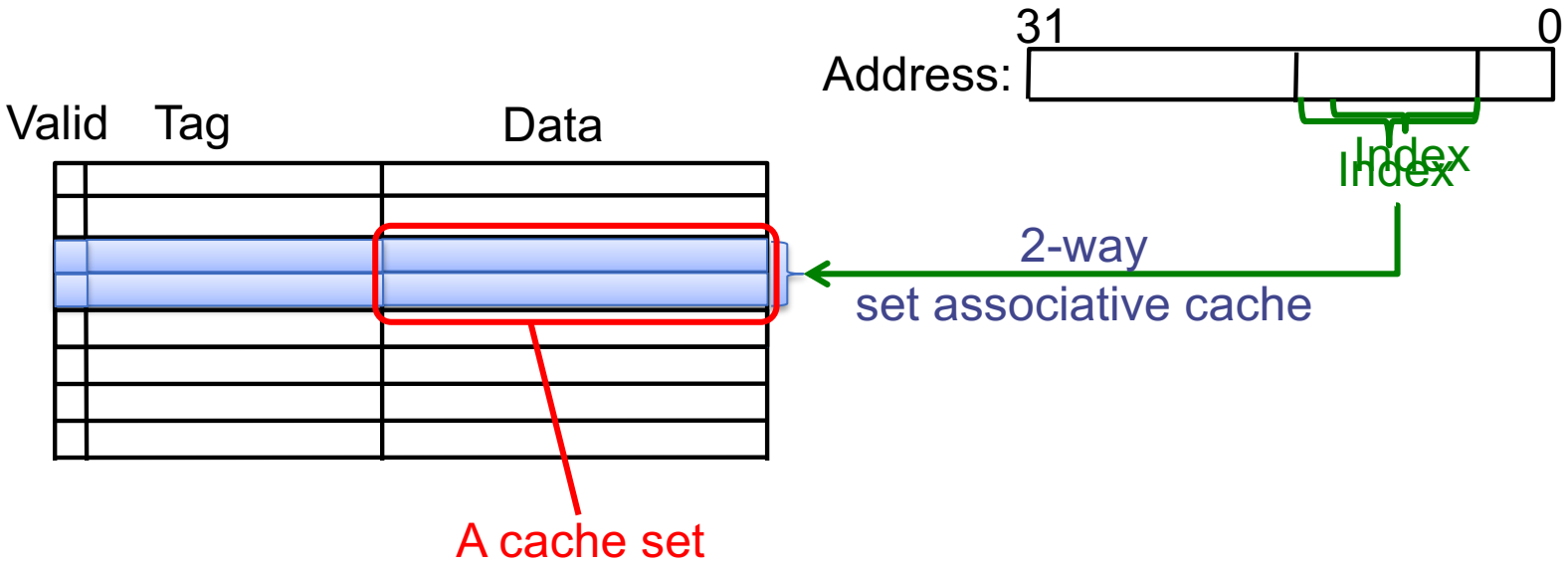


Directly vs. Associative Mapped Caching

- Direct mapped caching allows any given main memory block to be mapped into exactly one unique cache location.

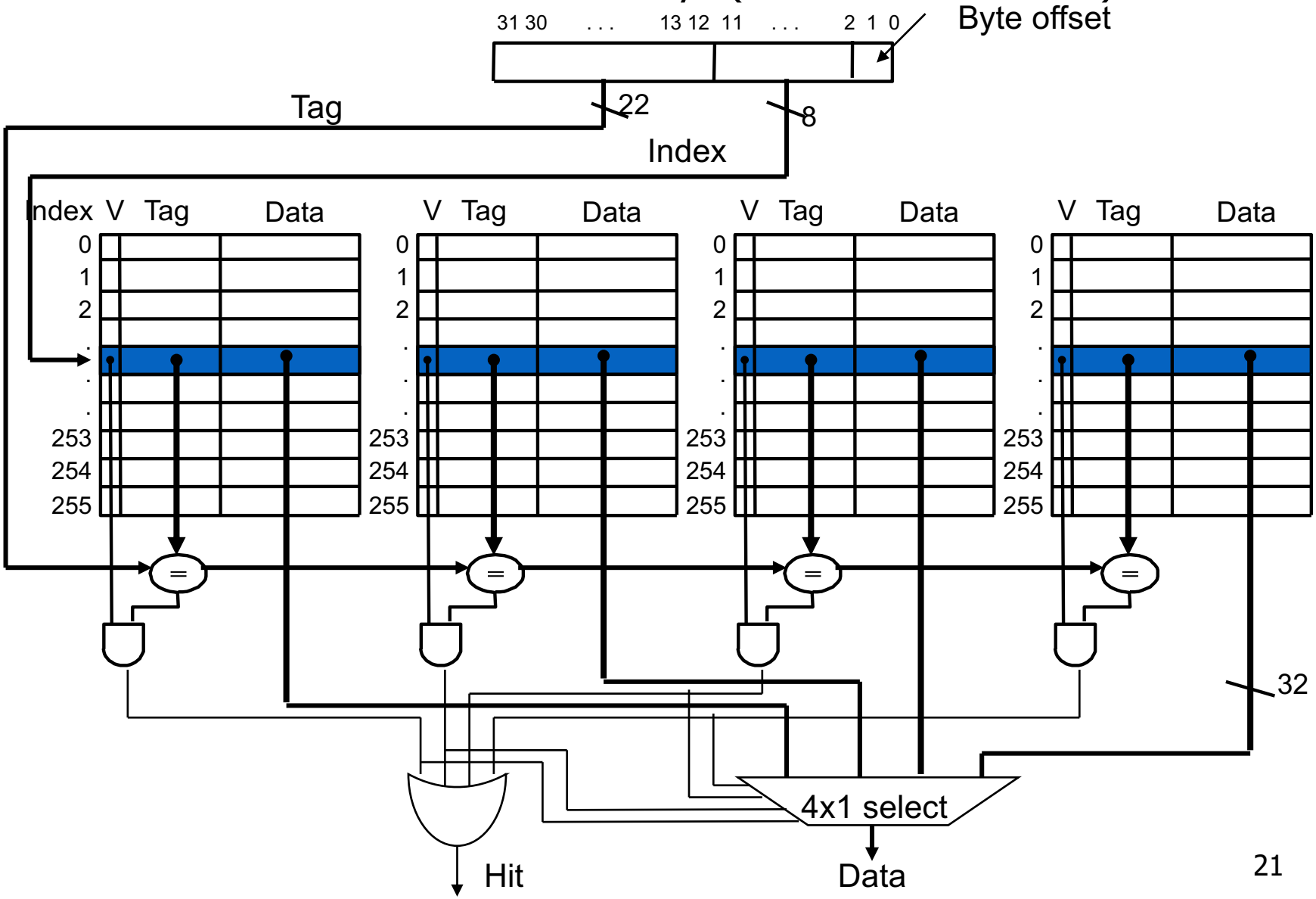
- Set-associative mapped cache allows any given main memory block to be mapped into two or more cache locations.

- Fully-associative mapped caching allows any given main memory block to be mapped into any cache location.



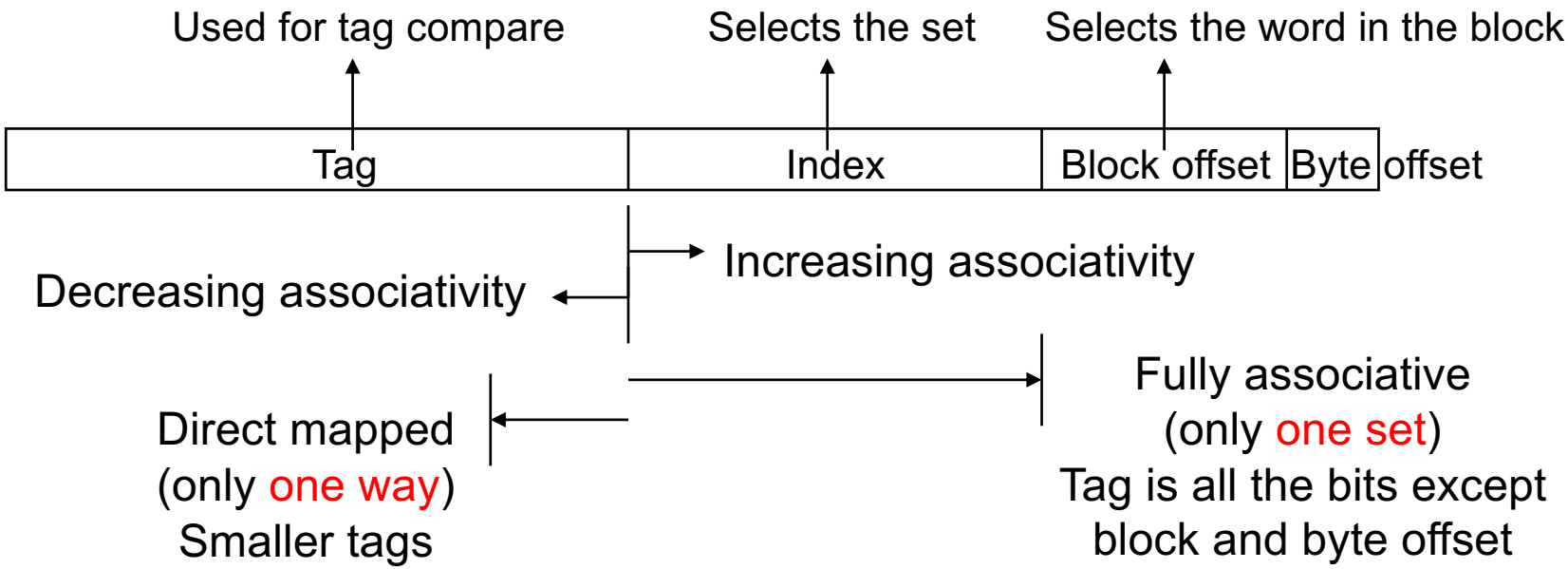
Example: 4-Way Set Associative Cache

- 2^8 = 256 sets each with four ways (each with one block)



Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



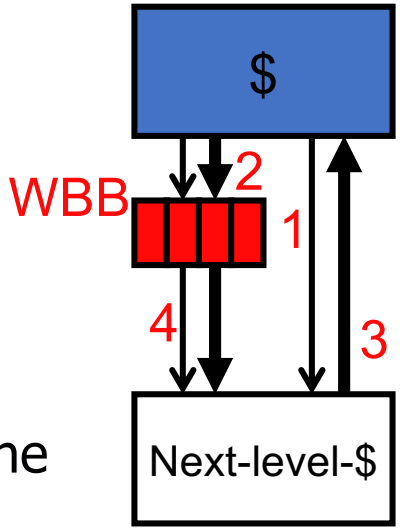
Handling Cache Hits and Misses

Handling Read Hits

- Read hits (I$ and D$)
  - this is what we want – no challenges

Handling Write Hits

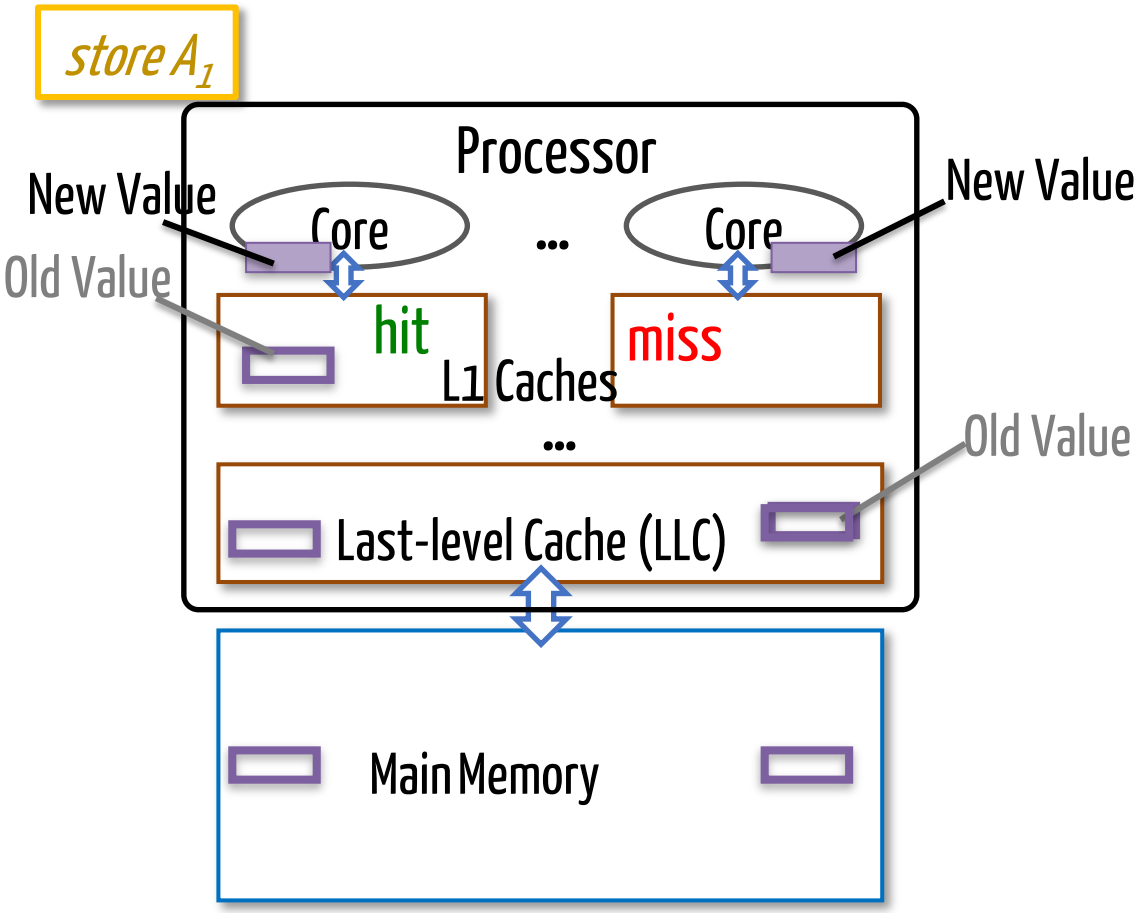
- When to propagate new value to (lower level) memory?
- **Option #1: Write-through**: consistent across hierarchy
  - Update the cache and immediately send the write to the next levels
  - Force consistent values across caches and main memory
- **Option #2: Write-back**: allow inconsistency across caches and main memory
  - Write the data only into the cache block : **no extra traffic**
  - write-back the cache contents to the next level in the memory hierarchy when that cache block is "evicted"
  - Requires additional "**dirty**" bit per 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



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

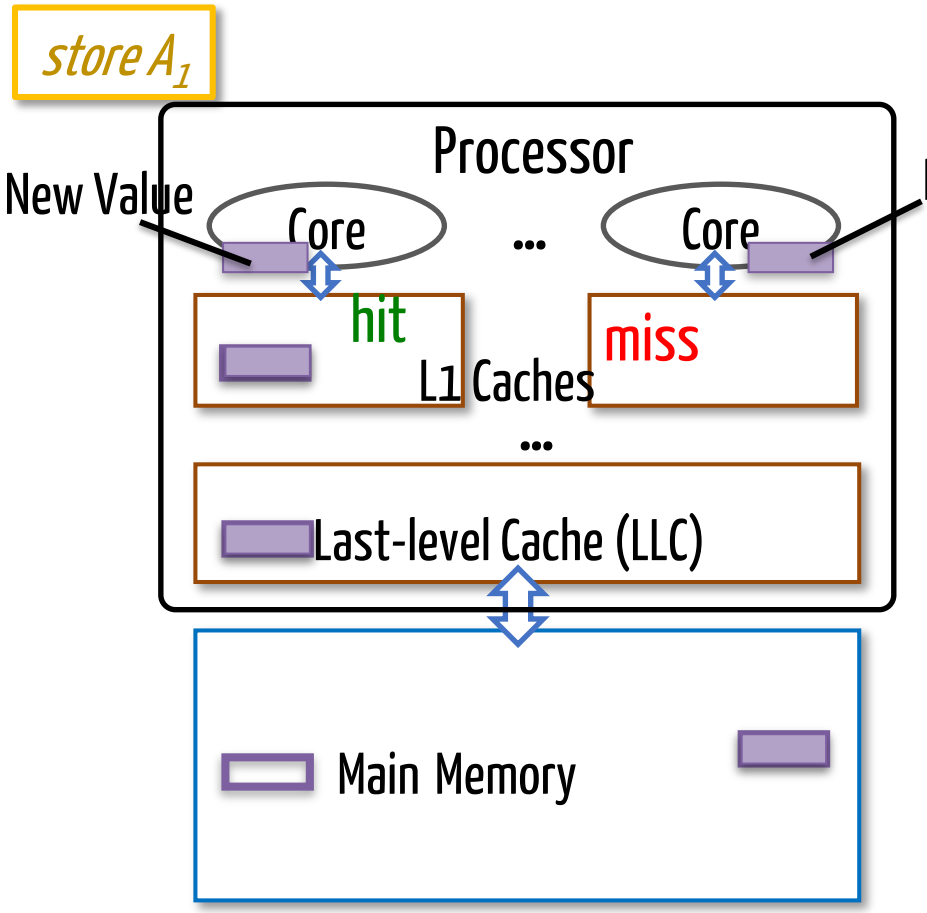
Handle writes in write-back cache



"Write-back Write-allocate"

- On write hit: write-back
- On write miss: write allocate, i.e., copy the old value from a lower level all the way up to L1

Handle writes in write-through cache

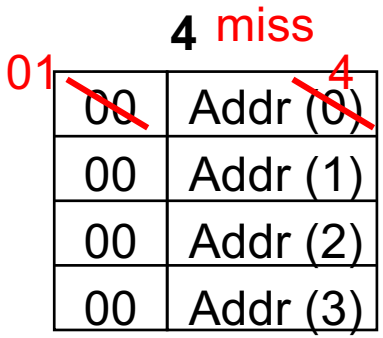


"Write-through Write-no-allocate"

- On write hit: write through
- On write miss: write-no-allocate, i.e., directly write to main memory, rather than bringing the cache block into caches (only bring cache blocks into caches on read misses)

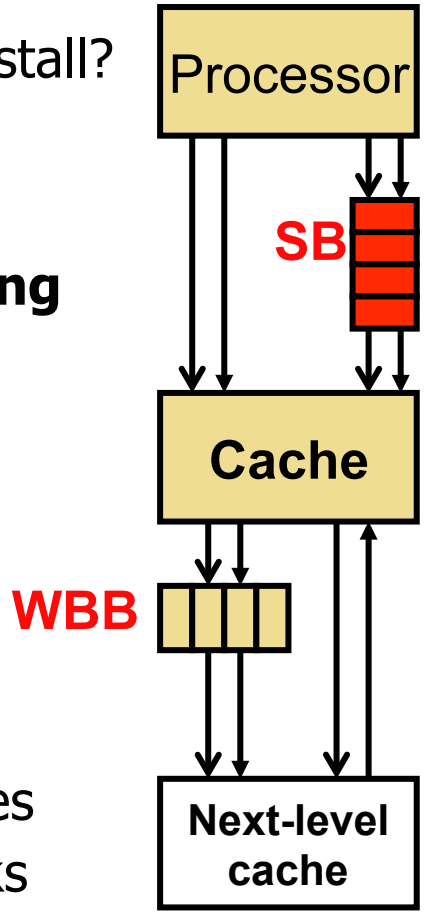
Handling a Cache Miss

- What if requested data isn't in the cache?
  - How does it get in there?
- **Cache controller** (a finite state machine)
  - Remembers miss address
  - Accesses next level of memory
  - Waits for response
  - Writes data/tag into proper locations
- What if we need to kick a cache block out?



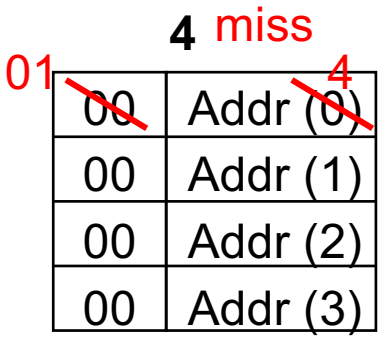
Cache misses and store buffers

- 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$)
  - \+ Eliminates stalls on write misses (mostly)
  - – Creates some problems in multiprocessors (later)
- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



Cache replacement in associative cache

- What if we need to kick a cache block out?
- There is only one option in directly-mapped cache



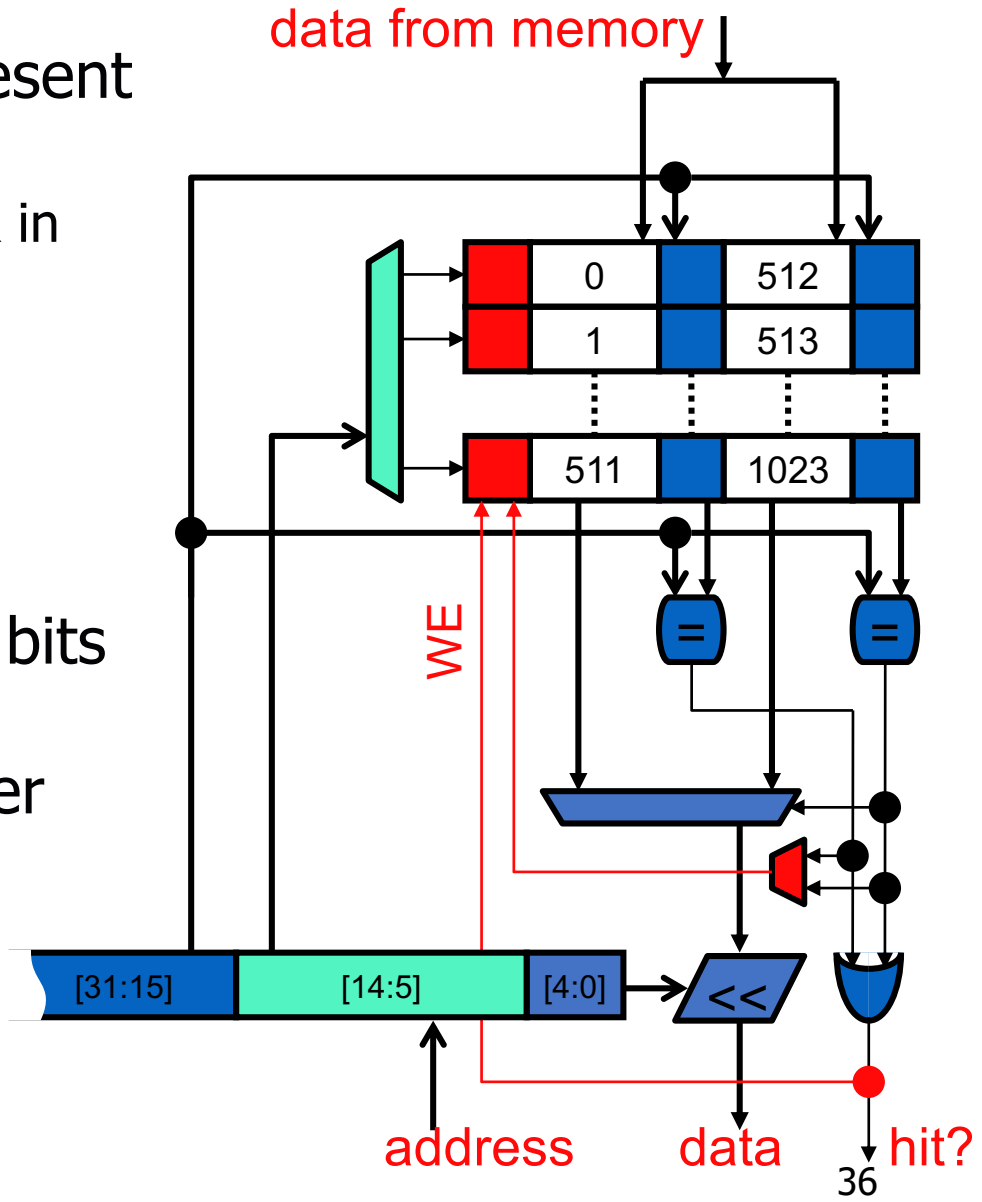
- What if set associative cache?

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

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



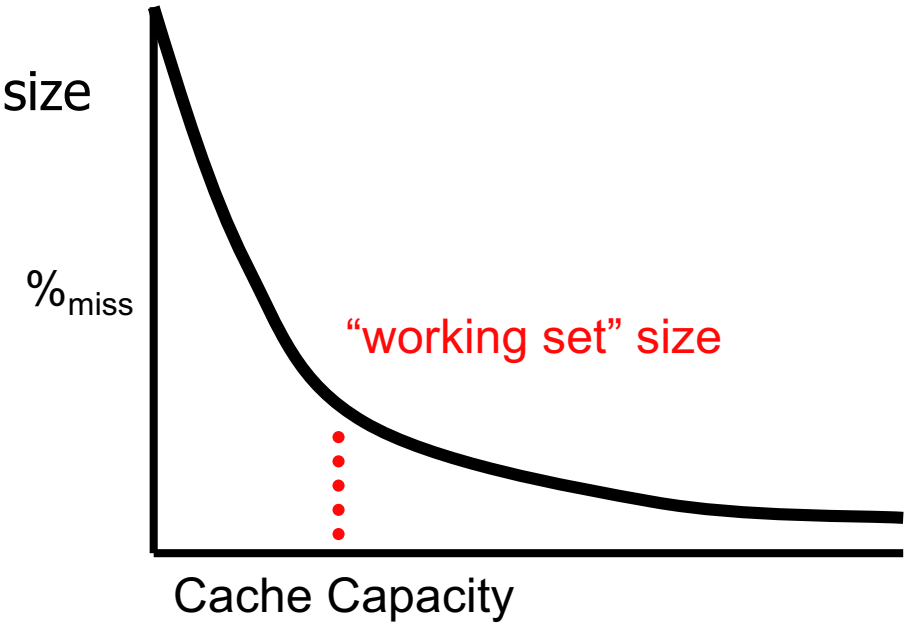
Classifying Misses: 3C Model

- 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)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

More about performance

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_{hit} increases
    - Latency grows with cache size
  - t_{avg} ?



Block Size

- For fixed capacity, reduce $\%_{miss}$ by changing organization
- One option: increase **block size** (an example on next slide)
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same

Previous example

- Consider the main memory word reference string

Start with an empty cache

block address: 0 1 2 3 4 3 4 15

0 miss

tag	
00	Mem(0)

Index: 0 mod 4 = 0

1 miss

00	Mem(0)
00	Mem(1)

1 mod 4 = 1

2 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)

2 mod 4 = 2

3 miss

00	Mem(0)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 mod 4 = 3

4 miss

01 ~~00~~	Mem(~~0~~) 4
00	Mem(1)
00	Mem(2)
00	Mem(3)

Index: 4 mod 4 = 0

3 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

3 mod 4 = 3

4 hit

01	Mem(4)
00	Mem(1)
00	Mem(2)
00	Mem(3)

4 mod 4 = 0

15 miss

01	Mem(4)
00	Mem(1)
00	Mem(2)
11 ~~00~~	Mem(~~3~~) 15

15 mod 4 = 3

- 8 requests, 6 misses

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache block address: 0 1 2 3 4 3 4 15

(Block address: 0000 0001 0010 0011 0100 0011 0100 1111)

tag

0 miss

00	Addr(1) Addr(0)

1 hit

00	Addr(1) Addr(0)

2 miss

00	Addr (1) Addr(0)
00	Addr (3) Addr(2)

3 hit

00	Addr (1) Addr(0)
00	Addr (3) Addr(2)

4 miss

~~01~~ 00	Addr ~~(1)~~ 5 Addr ~~(0)~~ 4
00	Addr (3) Addr(2)

3 hit

01	Addr (5) Addr(4)
00	Addr (3) Addr(2)

4 hit

01	Addr (5) Addr(4)
00	Addr (3) Addr(2)

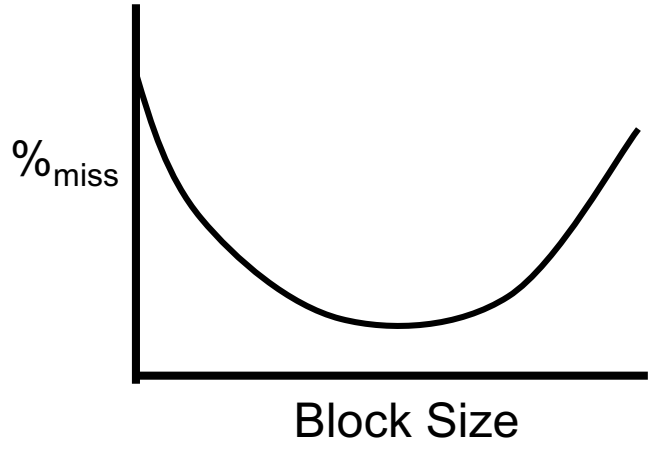
15 miss

11 01	Addr (5) Addr(4)
~~00~~	Addr ~~(3)~~ 15 Addr ~~(2)~~ 14

- 8 requests, 4 misses

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



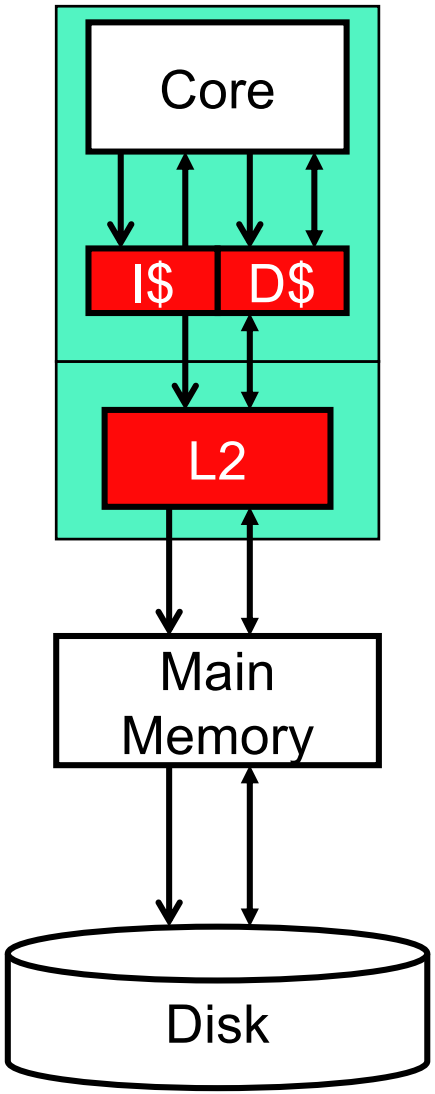
Miss Rates: per "access" vs "instruction"

- 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)

Summary: 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