CMPT 300
Introduction to Operating Systems

Cache

Acknowledgement: some slides are taken from CS61C course material at UC Berkeley

Agenda

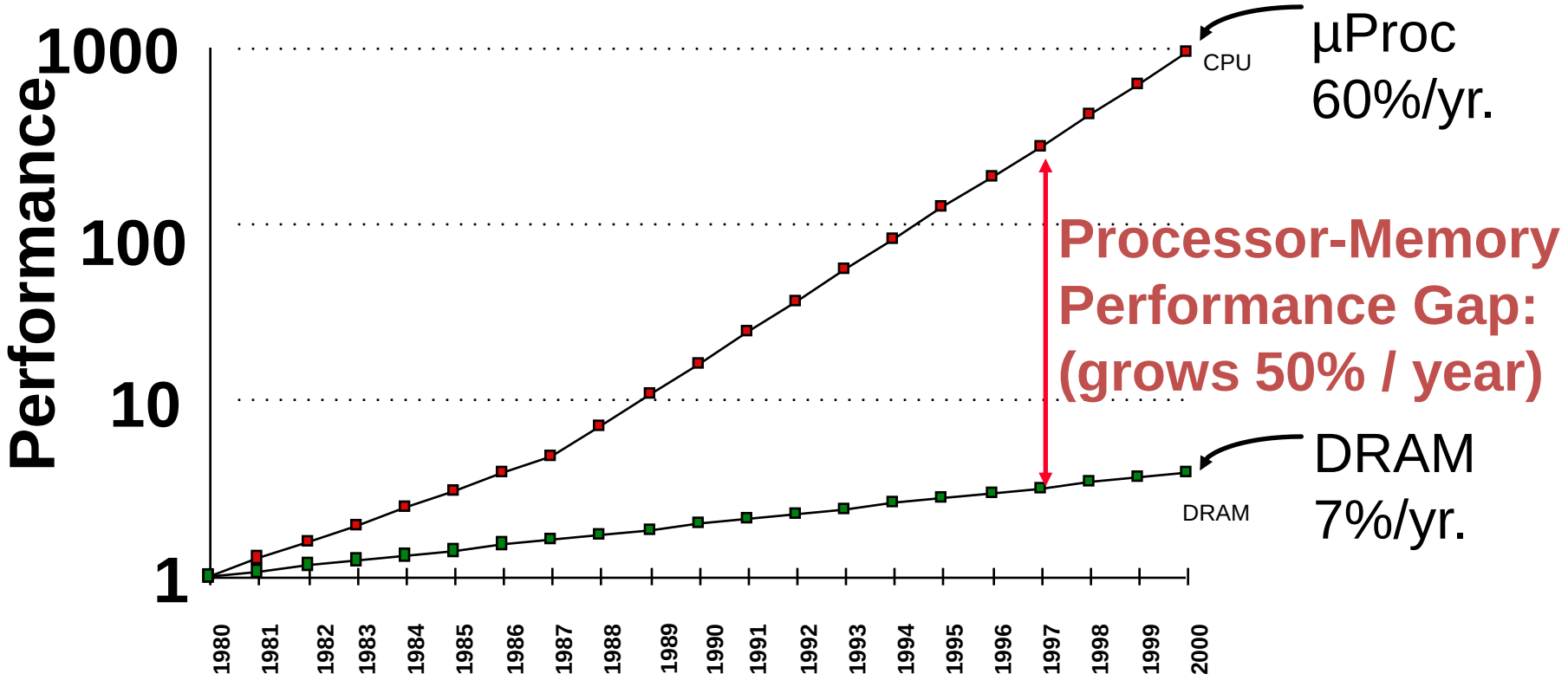
- Memory Hierarchy
- Direct Mapped Caches
- Cache Performance
- Set Associative Caches
- Multiprocessor Cache Consistency

Storage in a Computer

- Processor
  - holds data in register file (~100 Bytes)
  - Registers accessed on sub-nanosecond timescale
- Memory (we'll call "main memory")
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!

Historical Perspective

- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two cache levels on chip



Library Analogy

- Writing a report on a specific topic.
- While at library, check out books and keep them on desk.
- If need more, check them out and bring to desk.
  - But don't return earlier books since might need them
  - Limited space on desk; Which books to keep?
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of books in the library

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time

Locality

- *Temporal Locality* (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on the same topic since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Does Locality Exist?

- Instruction accesses spend a lot of time on the same page (since accesses mostly sequential, loop body tends to be small)
- Stack accesses have definite locality of reference
- Data accesses have less locality, but still some, depending on application…

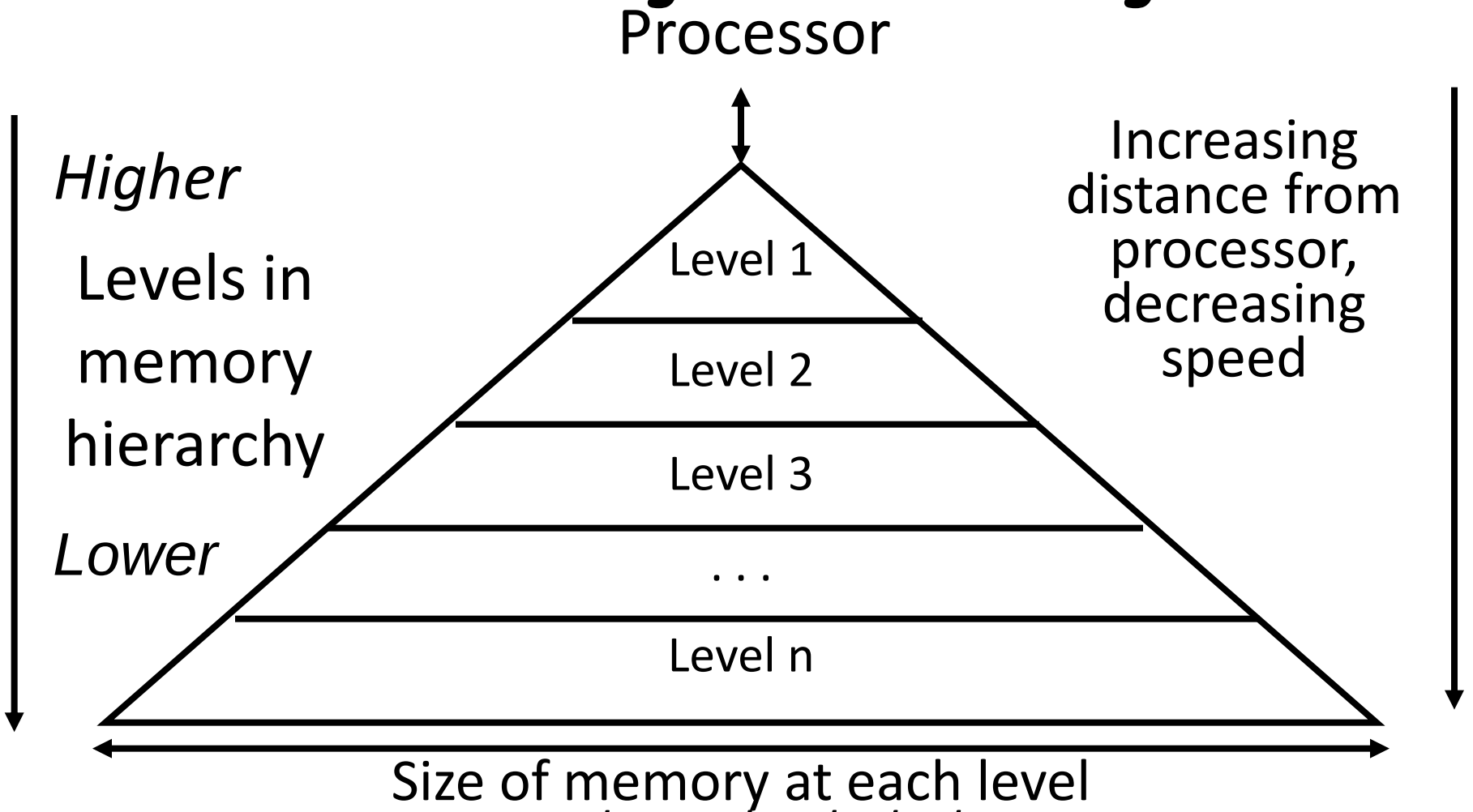
How does hardware exploit principle of locality?

- Offer a hierarchy of memories where
  - closest to processor is fastest (and most expensive per bit so smallest)
  - furthest from processor is largest (and least expensive per bit so slowest)
- Goal is to create illusion of memory almost as fast as fastest memory and almost as large as biggest memory of the hierarchy

Motivation for Mem Hierarchy

- Wanted: size of the largest memory available, speed of the fastest memory available
- Approach: Memory Hierarchy
  - Successively lower levels contain “most used” data from next higher level
  - Exploits *temporal & spatial locality*

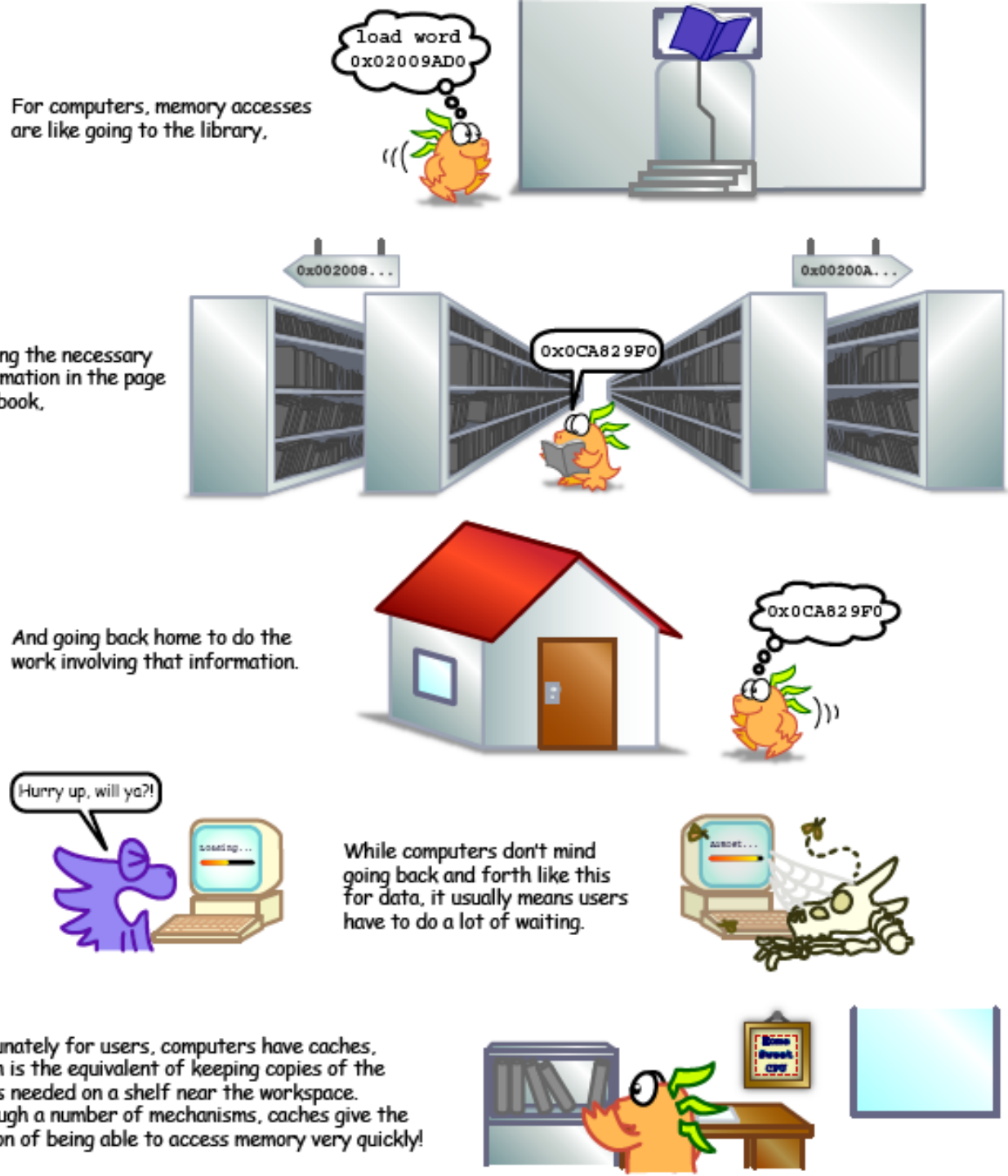
Memory Hierarchy



As we move to deeper levels the latency goes up and price per bit goes down.

Cache

- Processor and memory speed mismatch leads us to add a new level: a *cache*
- Implemented with same integrated circuit processing technology as processor, integrated on-chip: faster but more expensive than DRAM memory
- Modern processors have separate caches for instructions and data, as well as several levels of caches implemented in different sizes
- As a pun, often use $ (“cash”) to abbreviate cache, e.g. D$ = Data Cache, I$ = Instruction Cache
- *Cache is a copy of a subset of main memory*

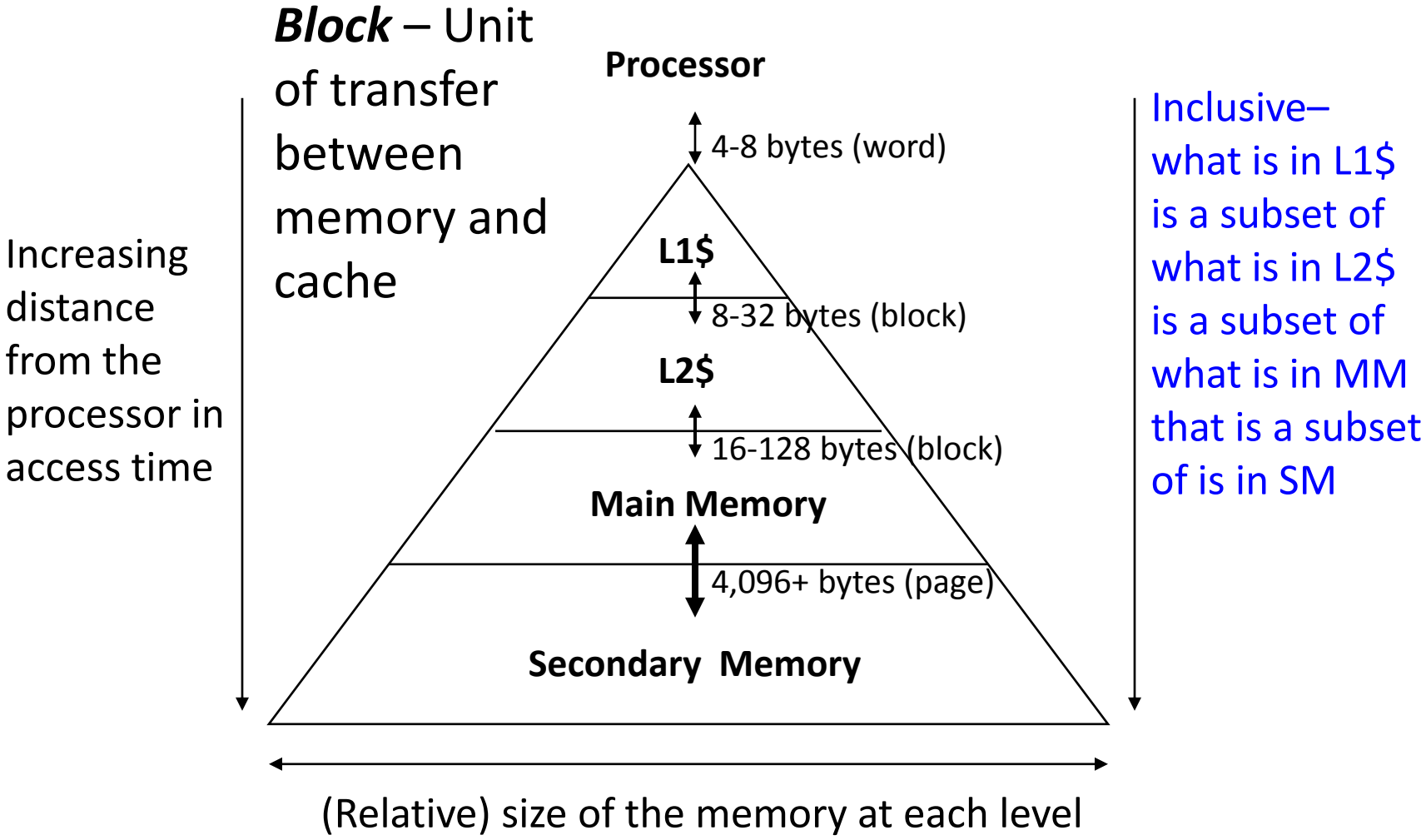


- Illustration from http://csillustrated.berkeley.edu/PDFs/cache-basics.pdf

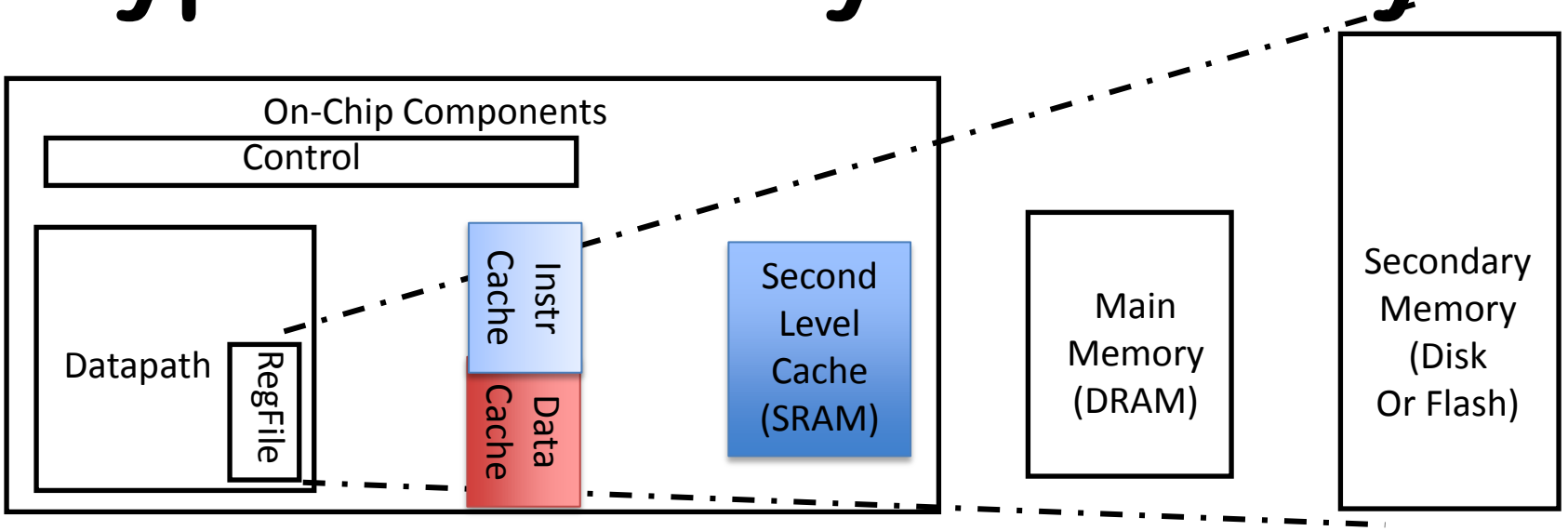
Memory Hierarchy Technologies

- Caches use SRAM (Static RAM) for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), higher power, expensive
    ($2000 to $4000 per GB in 2011)
  - Static: content will last as long as power is on
- Main memory uses DRAM (Dynamic RAM) for size (density)
  - Slower (typical access times of 50 to 70 ns)
  - High density (1 transistor cells), lower power, cheaper
    ($20 to $40 per GB in 2011)
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - Consumes 1% to 2% of the active cycles of the DRAM

Characteristics of the Memory Hierarchy



Typical Memory Hierarchy



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	←		→	lowest

- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

How is the Hierarchy Managed?

- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware
- main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
    - Virtual to physical address mapping assisted by the hardware (TLB)
  - By the programmer (files)

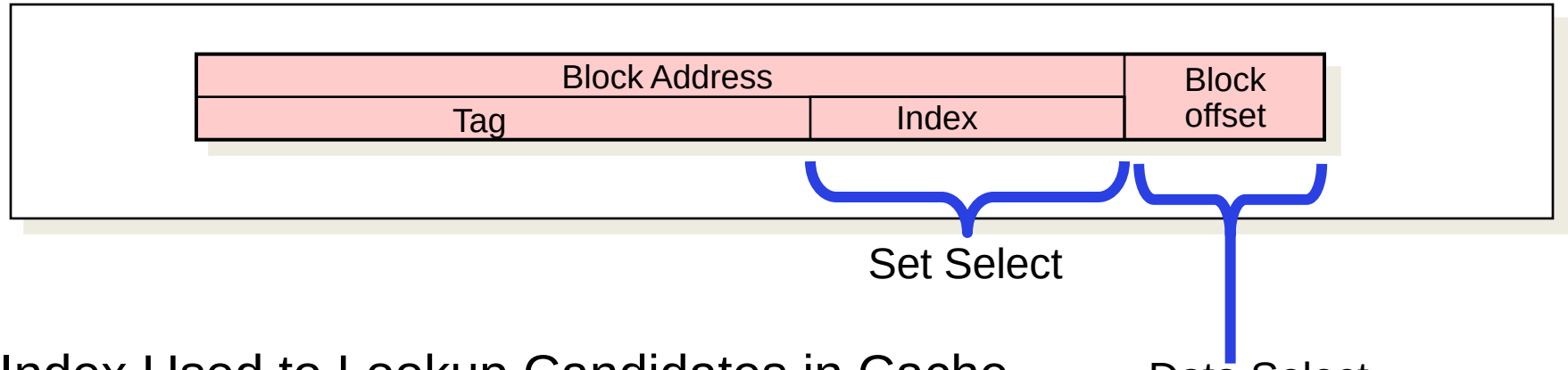
Cache Management

- Cache managed automatically by hardware.
- Operations available in hardware are limited, scheme needs to be relatively simple.
- Where in the cache do we put a block of data from memory?
  - How do we find it when we need it?
- What is the overall organization of blocks we impose on our cache?

Direct Mapped Caches

- Each memory block is mapped to exactly one block in the cache
  - A “cache block” is also called a “cache line”
  - Only need to check this single location to see if block is in cache.
- Cache is smaller than memory
  - Multiple blocks in memory map to a single block in the cache!
  - Need some way of determining the identity of the block.

Direct Mapped Cache

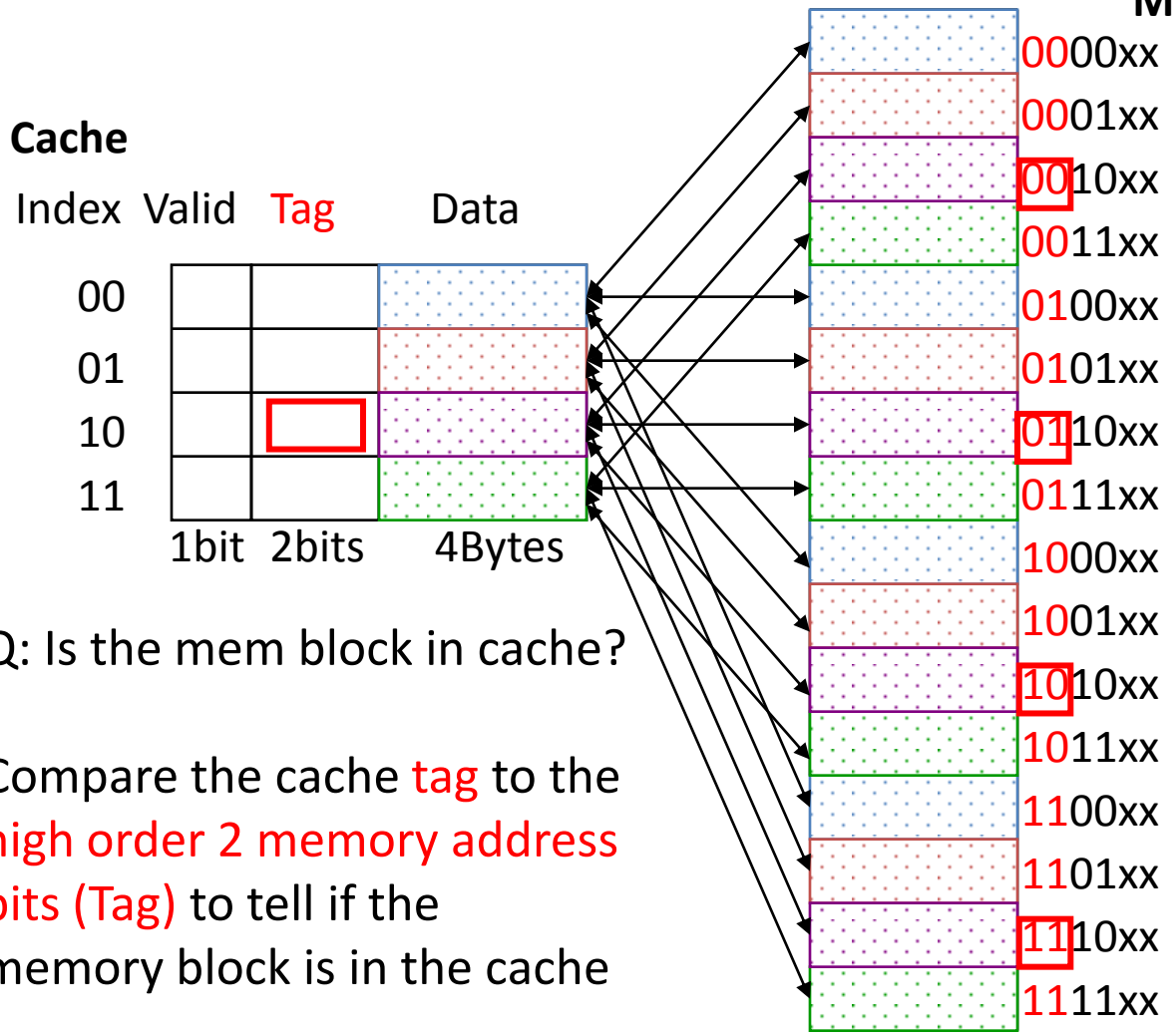


- Index Used to Lookup Candidates in Cache
  - Index identifies the set; *which address in the cache* is the block stored?
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss; since multiple memory addresses can map to the same cache address, *which block in memory* did the block come from?
- Block is minimum quantum of caching
  - Offset field used to select data within block; *which byte within a block* is referenced?
- Address mapping:
  - cache index = (memory *block* address (Tag&Index)) modulo (# of *blocks* in the cache)

Summary: Tag, Index, Offset

- Tag
  - used to distinguish between multiple memory addresses that map to a given cache block index
- Index
  - specifies the cache block index (which “row”/block of the cache we should look in)
  - I bits <=> 2^I blocks in cache
- Offset
  - once we’ve found correct cache block index, specifies which byte within the block we want (which “column” in the cache)
  - O bits <=> 2^O bytes per block
- Each block in memory maps to one block in the cache.
  - Index to determine which block
  - Tag to determine if it’s the right block
  - Offset to determine which byte within block

Direct Mapped Cache Example



Tag: upper 2 bits → 4 memory addresses mapped to the same cache block index

Index: middle 2 bits → 4 blocks in cache; Index defines which cache block index the mem address is mapped to, by modulo 4 arithmetic

Offset: lower 2 bits → 1 word (4 bytes) per block; Offset defines the byte within the cache block

Q: Is the mem block in cache?

Compare the cache tag to the high order 2 memory address bits (Tag) to tell if the memory block is in the cache

Index = (block address) modulo (# of blocks in the cache (4))

Quiz

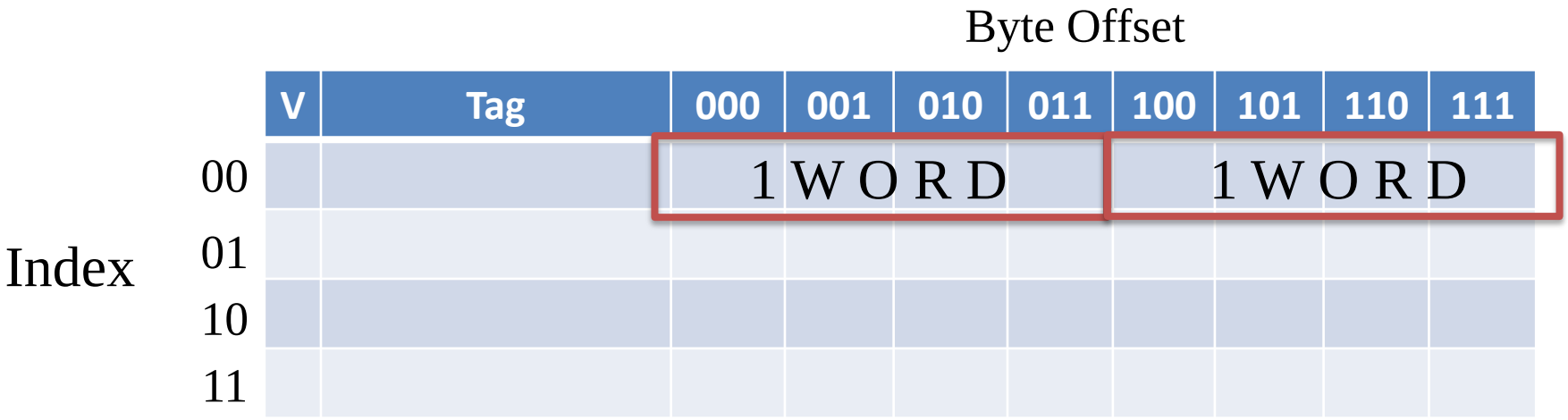
- What if we flip positions of Tag and Index? Have Index as higher-order bits, and Tag as lower-order bits?
  - Exercise: redraw the previous slide with this configuration
- Bad idea: Neighboring blocks in memory are mapped to the same cache line, since they share the same Index value;
  - if memory blocks are accessed sequentially, then each new incoming cache block will immediately replace the previous block at the same cache line, causing cache thrashing → cannot exploit spatial locality.

Word vs. Byte

- Most modern CPUs, e.g., MIPS, operate with words (4 bytes), since it is convenient for 32-bit arithmetic. Entire words will be transferred to and from the CPU. Hence the only possible byte offsets are multiples of 4.
- When we use words as the unit of memory size, we leave out the last 2 bits of offset for byte addressing

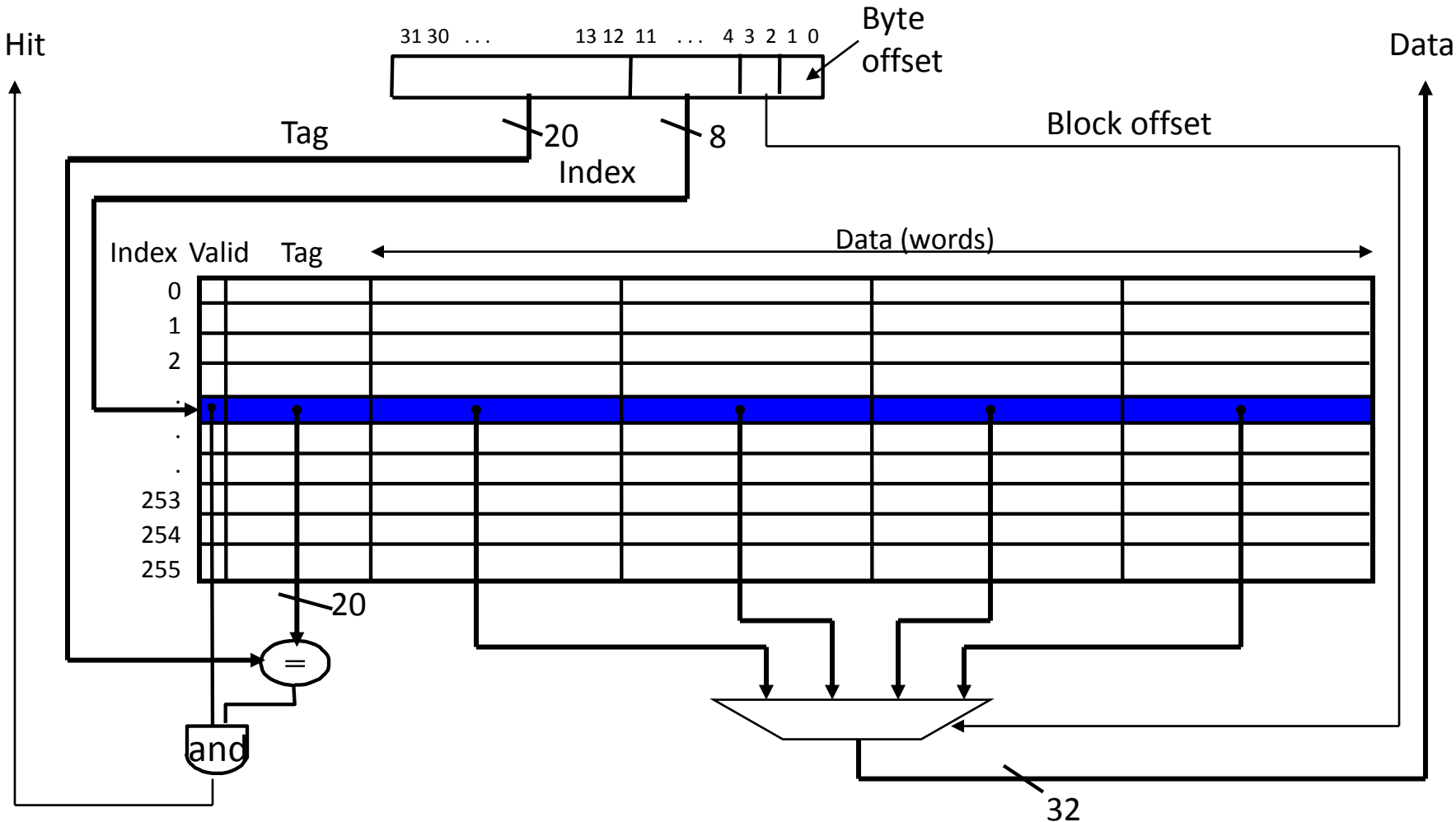
Direct Mapped Cache Example

- 8-bit address space. Tag: 3 bits, Index: 2 bits, Offset: 3 bits
- Index 2 bits → 2^2=4 cache blocks
- Offset 3 bits → 8 Bytes/block,
  - Cache size = 4 blocks * 8 Bytes/block = 32 Bytes (8 words)
- Tag 3 bits → 2^3=8 memory addresses mapped to the same cache index



Direct Mapped Cache Example 2

- 32 bit address space. Tag: 20 bits; Index: 8 bits; Offset: 4 bits
- 2^8=256 cache blocks, 2^4=16 Bytes/block
  - Cache size = 256 blocks * 16 Bytes/block = 4K Bytes (1K words)
- 2^20=1 million memory addresses mapped to each cache index



Caching Terminology

- When accessing a memory address, 2 things can happen:
  - cache hit:
    cache block is valid and refers to the proper memory address, so read desired word from cache (fast)
  - cache miss:
    cache block is invalid, or refers to the wrong memory address, so read from memory (slow)

Direct Mapped Cache Example

- Consider the 6-bit memory address example; sequence of memory address accesses (Byte offset bits are ignored, so only 4-bit word addresses: Tag 2bits; Index 2bits)

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0000 0001 0010 0011 0100 0011 0100 1111

0 miss

00	Mem(0)

1 miss

00	Mem(0)
00	Mem(1)

2 miss

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

3 miss

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

Time →

4 miss

01 4

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

3 hit

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

4 hit

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

15 miss

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

11 15

Time →

- 8 requests, 6 misses

Taking Advantage of Spatial Locality

- Each cache block holds 2 words; so Tag 2bits; Index 1bit; Offset 1bit (for 1 of 2 words in block)

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

0000 0001 0010 0011 0100 0011 0100 1111

0 miss

00	Mem(1)	Mem(0)

1 hit

00	Mem(1)	Mem(0)

2 miss

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

3 hit

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

4 miss

01

~~00~~	Mem(1) 5	~~Mem(0)~~ 4
00	Mem(3)	Mem(2)

3 hit

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

4 hit

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

15 miss

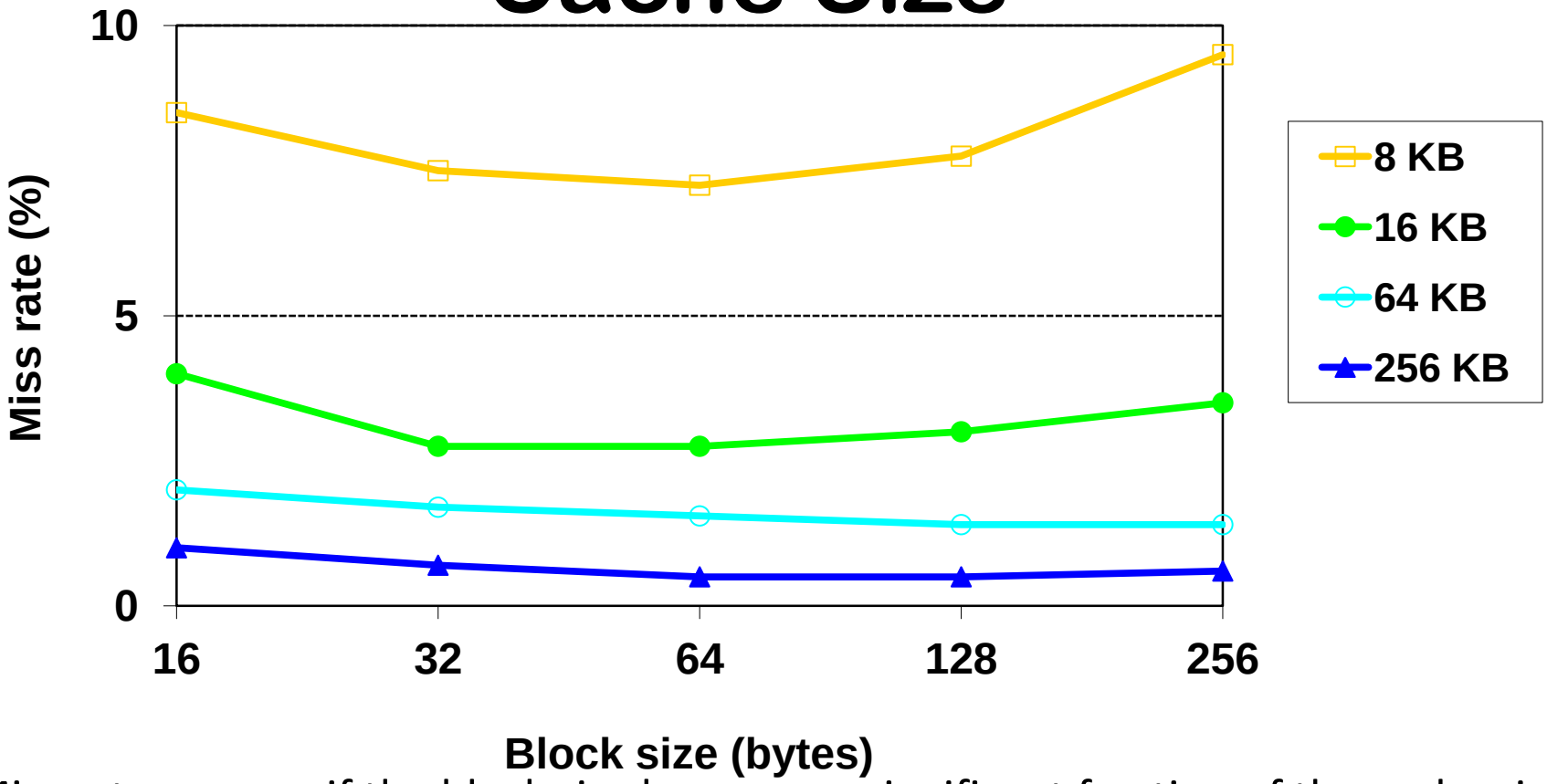
11

01	Mem(5)	Mem(4)
~~00~~	Mem(3)	~~Mem(2)~~

15 14

- 8 requests, 4 misses

Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)
- A more serious problem is that the miss penalty goes up, since it is the time to fetch the block from the next lower level of the hierarchy and load it into the cache.

Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

  AMAT = Hit Time + Miss rate x Miss penalty

- Q:Why is it not "(1-Miss rate) x Hit Time + Miss rate * Miss Time"?
- A: Same, since Miss Time=Hit Time + Miss penalty
- What is the AMAT for a processor with a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

  1 + 0.02 x 50 = 2 clock cycles

  Or 2 x 200 = 400 psecs

Potential impact of much larger cache on AMAT?

- 1) Lower Miss rate
- 2) Longer Access time (Hit time): smaller is faster
  - Increase in hit time will likely add another stage to the CPU pipeline
- At some point, increase in hit time for a larger cache may overshadow the improvement in hit rate, yielding a overall decrease in performance

Measuring Cache Performance – Effect on CPI

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

  CPU time = IC (Instruction Count) × CPI (Cycles per Instruction)× CC (Clock Cycle)

  = IC × (CPI_{ideal} + Average Memory-stall cycles) × CC

  (CPI_{stall})

- A simple model for Memory-stall cycles

Memory-stall cycles = accesses/instruction × miss rate × miss penalty

Impacts of Cache Performance

- Relative $ penalty increases as processor performance improves (faster clock rate and/or lower CPI)
  - Memory speed unlikely to improve as fast as processor cycle time. When calculating CPI_{stall}, cache miss penalty is measured in processor clock cycles needed to handle a miss
  - Lower the CPI_{ideal}, more pronounced impact of stalls
- Processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I$ and 4% D$ miss rates
  - Memory-stall cycles = 2% × 100 + 36% × 4% × 100 = 3.44
  - So $CPI_{stalls} = 2 + 3.44 = 5.44$
  - More than twice CPI_{ideal}!

Handling Cache Misses (Single Word Blocks)

- Read misses (I$ and D$)
  - Stall execution, fetch the block from the next level in the memory hierarchy, install it in the cache, send requested word to processor, and then let execution resume
- Write misses (D$ only)
  - Write allocate: fetch the block from memory to cache before writing the word from processor to cache

  or

  - No-write allocate: skip the cache write and just write the word to memory

Write-Allocate

- Consider a direct-mapped cache with 2-word (8-byte) cache block size. Assume we do a 4-byte write to memory location 0x000000 and causes a cache miss. Do we have to load the entire block (Bytes 0-7) from memory to cache before we write the cache tag and the 4-byte data into Bytes 0-3?
- If we do load the block before writing, it is called write allocate.
- If not, it is called write-no-allocate
  - In this case, you need some way to tell the processor the rest of the block is no longer valid.
  - Otherwise, you may have half of the cache block (Bytes 0-3) referring to one memory block, and the other half (Bytes 4-7) referring to another memory block, both mapped to the same cache index!

Cache-Memory Consistency? (1/2)

- Need to make sure cache and memory are consistent (know about all updates)

1) Write-Through Policy: write cache and write *through* the cache to memory
   - Every write eventually gets to memory
   - Too slow, so include Write Buffer to allow processor to continue once data in Buffer, Buffer updates memory in parallel to processor

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache
   - Writes collected in cache, only single write to memory per block
   - Include bit to see if wrote to block or not, and then only write back if bit is set
     - Called "Dirty" bit (writing makes it "dirty")

- Typical combinations:
- Write-back + write allocate,
  - Hoping for a subsequent writes (or reads) to the same location, which is now cached.
- Write-through + no-write allocate.
  - Consecutive writes have no advantage, since they still need to be written directly to memory

Write-through vs. Write-back

- WT:
  - PRO: read misses cannot result in writes
  - CON: Processor held up on writes unless writes buffered
- WB:
  - PRO: repeated writes not sent to DRAM
    processor not held up on writes
  - CON: More complex
    Read miss may require write back of dirty block
    - Clean miss: the block being replaced is clean
    - Dirty miss: the block being replaced is dirty and needs to be written back to memory

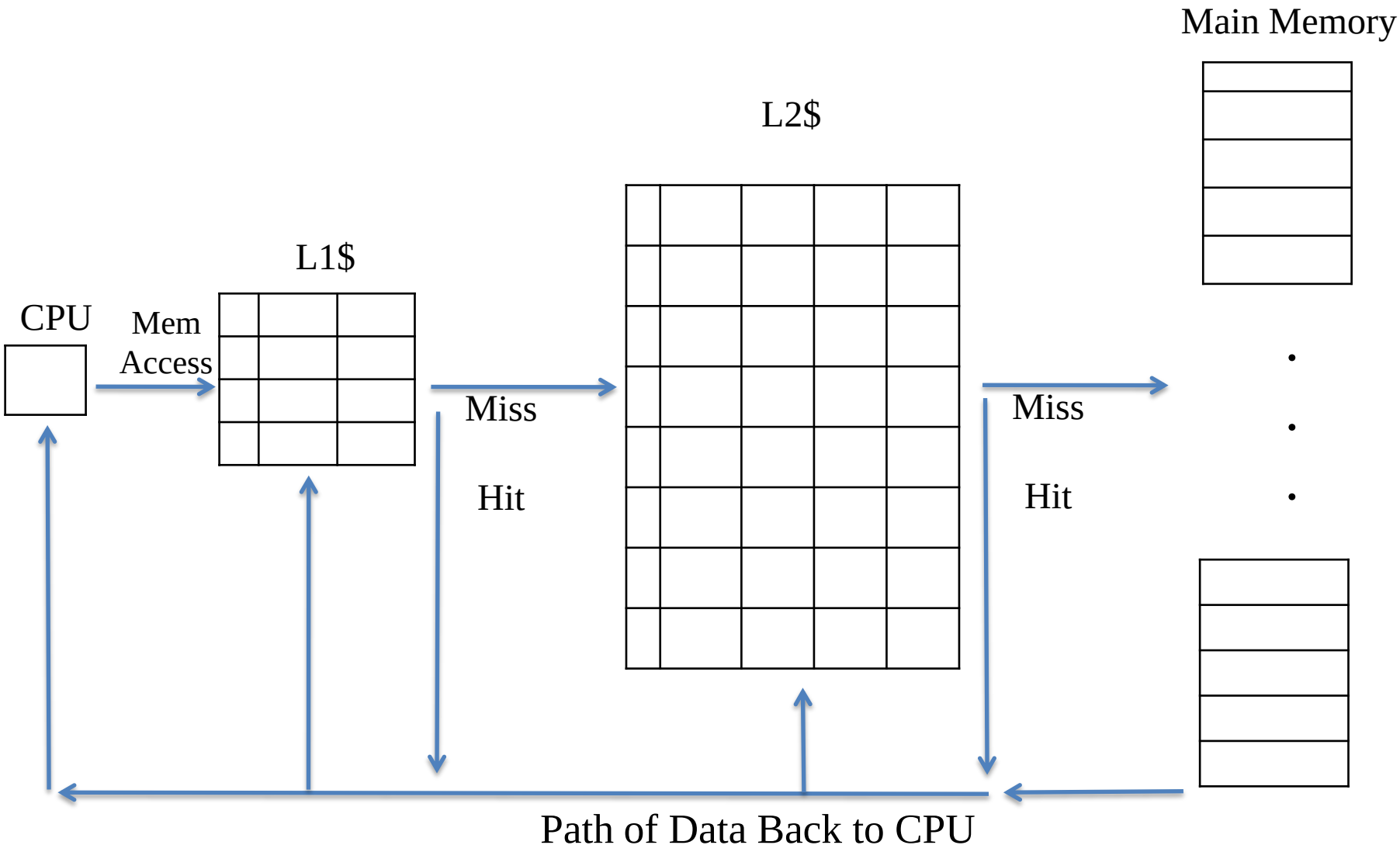
Recall: Average Memory Access Time (AMAT)

- Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

AMAT = Time for a hit + Miss rate x Miss penalty

- How reduce Miss Penalty?

Multiple Cache Levels



Multiple Cache Levels

- With advancing technology, have more room on die for bigger L1 caches and for second level cache – normally a unified L2 cache (i.e., it holds both instructions and data,) and in some cases even a unified L3 cache

- New AMAT Calculation:

  AMAT = L1 Hit Time + L1 Miss Rate * **L1 Miss Penalty**
  L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

  and so forth (final miss penalty is Main Memory access time)

New AMAT Example

- 1 cycle L1 Hit Time, 2% L1 Miss Rate, 5 cycle L2 Hit Time, 5% L2 Miss Rate.
- 100 cycle Main Memory access time
- No L2 Cache:

  AMAT = 1 + .02*100 = 3
- With L2 Cache:

  AMAT = 1 + .02*(5 + .05*100) = 1.2!

Local vs. Global Miss Rates

- ***Local miss rate*** – the fraction of references to one level of a cache that miss
  - Local Miss rate L2$ = $L2 Misses / L1$ Misses
- ***Global miss rate*** – the fraction of references that miss out of all memory accesses in system.
  - L2$ local miss rate >> than the global miss rate
- Global Miss rate = L2$ Misses / Total Accesses

  = L2$ Misses / L1$ Misses x L1$ Misses / Total Accesses

  = Local Miss rate L2$ x Local Miss rate L1$
- AMAT Calculation used **Local Miss Rate**.

Design Considerations

- Different design considerations for L1$ and L2$
  - L1$ focuses on fast access: minimize hit time to achieve shorter clock cycle, e.g., smaller $
  - L2$ (and L3$) focus on low miss rate: reduce penalty of long main memory access times: e.g., Larger $ with larger block sizes/higher levels of associativity
- Miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster even with higher miss rate
- For the L2$, fast hit time is less important than low miss rate
  - L2$ hit time determines L1$'s miss penalty
  - L2$ local miss rate >> than the global miss rate

Sources of Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - **Solution 2: increase associativity (may increase access time)**

Compulsory

Compulsory misses happen when a block is referenced for the first time. The computer can't get a block that doesn't exist yet!

Capacity

The block is not in the cache because there is no space in the cache for it. Caches are of finite size, after all.

Conflict

These types of misses happen only in direct-mapped and set-associative caches. Multiple blocks can be mapped to a set, forcing evictions when the set is full.

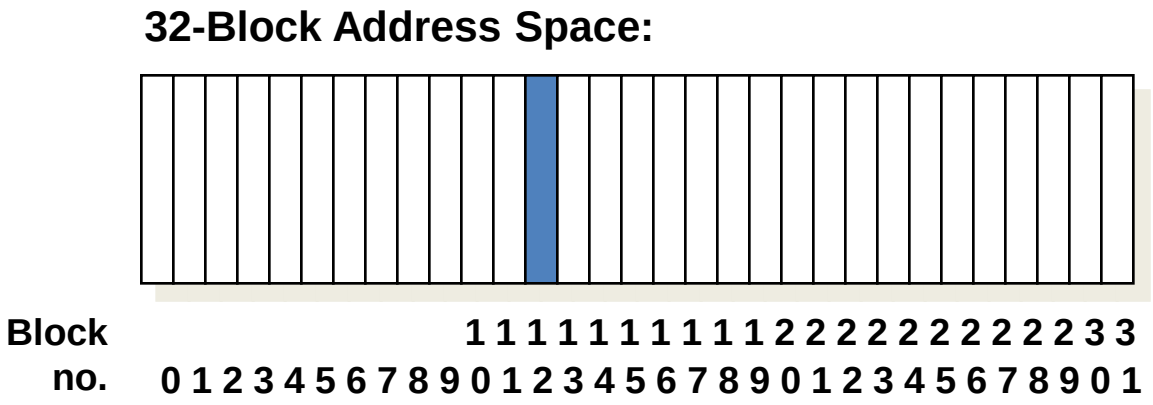
- Illustration from http://csillustrated.berkeley.edu/PDFs/cache-misses.pdf

Reducing Cache Misses

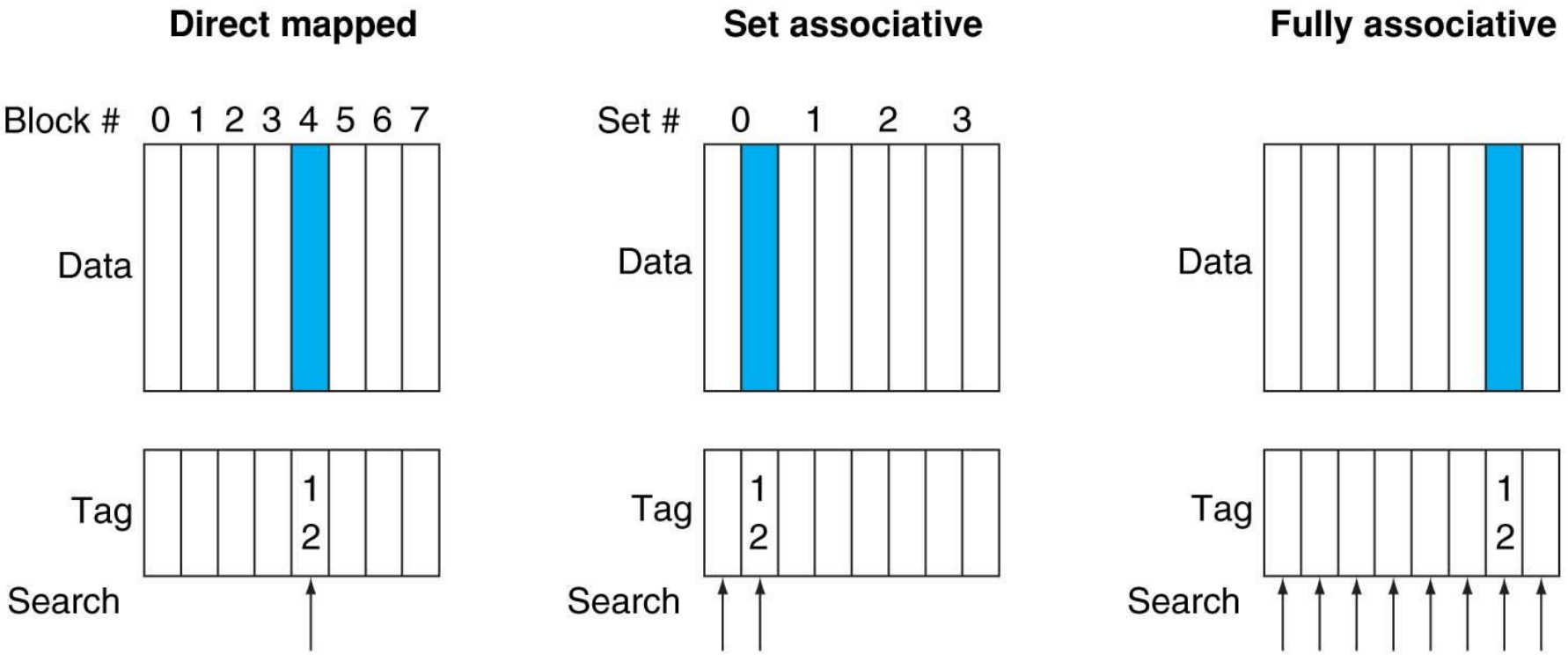
- Allow more flexible block placement in cache
- *Direct mapped $*: memory block maps to exactly one cache block
- *Fully associative $*: allow a memory block to be mapped to any cache block
- Compromise: divide $ into sets, each of which consists of n addresses (“ways”) (*n-way set associative*) to place memory block
  - Cache set index=(block address) modulo (# sets in the cache)
  - A block can be placed in any of the n addresses (ways) of the cache set it belongs to.

Alternative Block Placement Schemes

- Example: Block 12 in 5-bit (32-block) memory address space placed in 3-bit (8 block) cache address space.

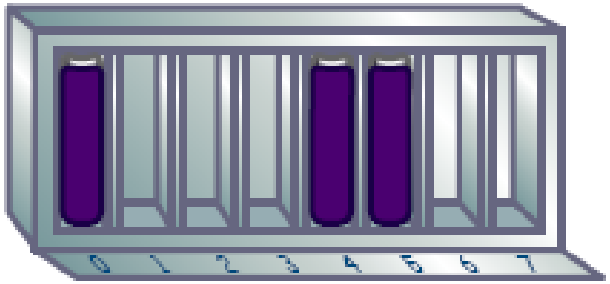


Alternative Block Placement Schemes



- DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- FA placement: mem block 12 can appear in any cache block

Direct Mapped



Tag	Index	Offset

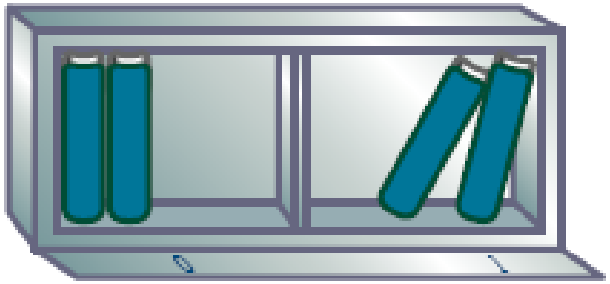
A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

2-Way Set Associative

Tag	Index	Offset

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

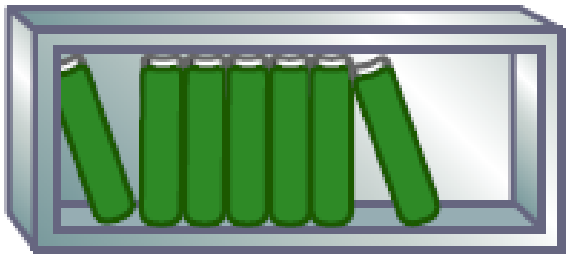
4-Way Set Associative



Tag	Index	Offset

Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

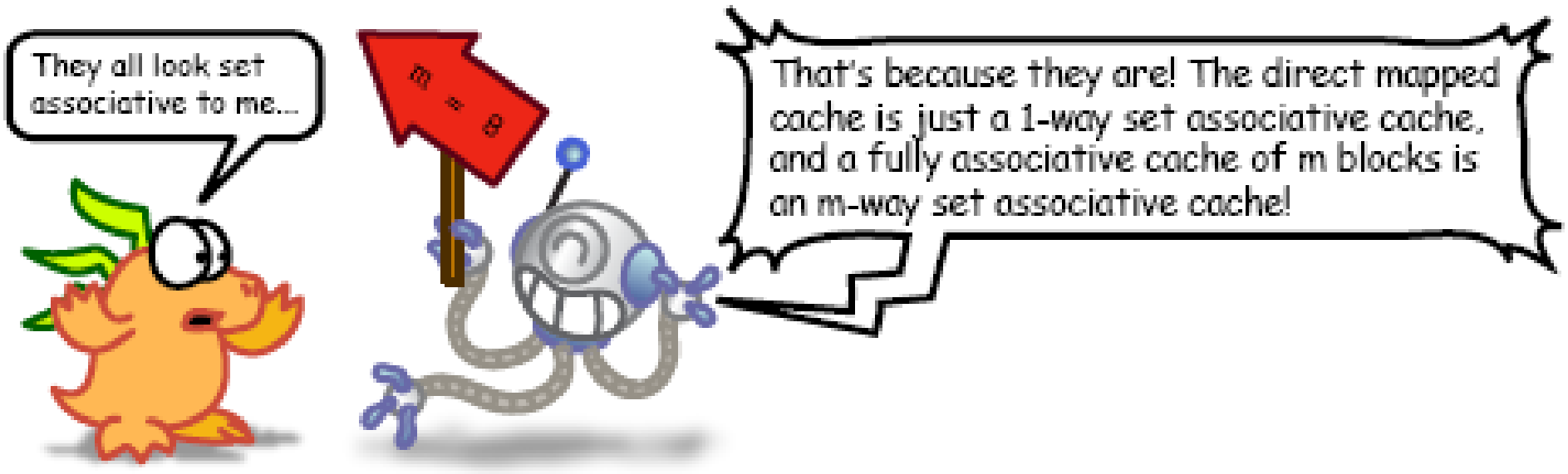
Fully Associative



Tag	Offset

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!

- Illustration from http://csillustrated.berkeley.edu/PDFs/cache-types.pdf

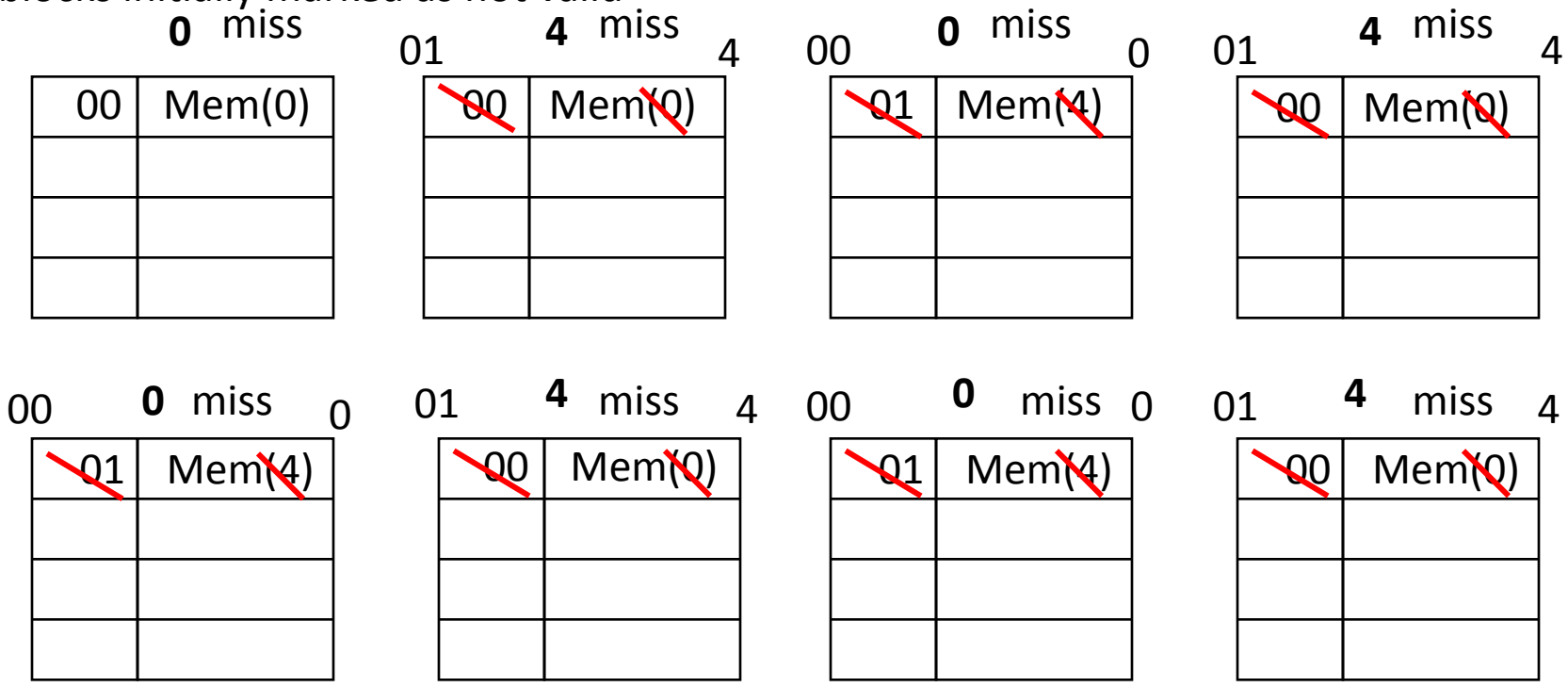

They all look set associative to me...
m = 8
That's because they are! The direct mapped cache is just a 1-way set associative cache, and a fully associative cache of m blocks is an m-way set associative cache!

Example: 4-Word Direct-Mapped $Worst-Case Reference String

- Consider the sequence of memory accesses

Start with an empty cache - all blocks initially marked as not valid

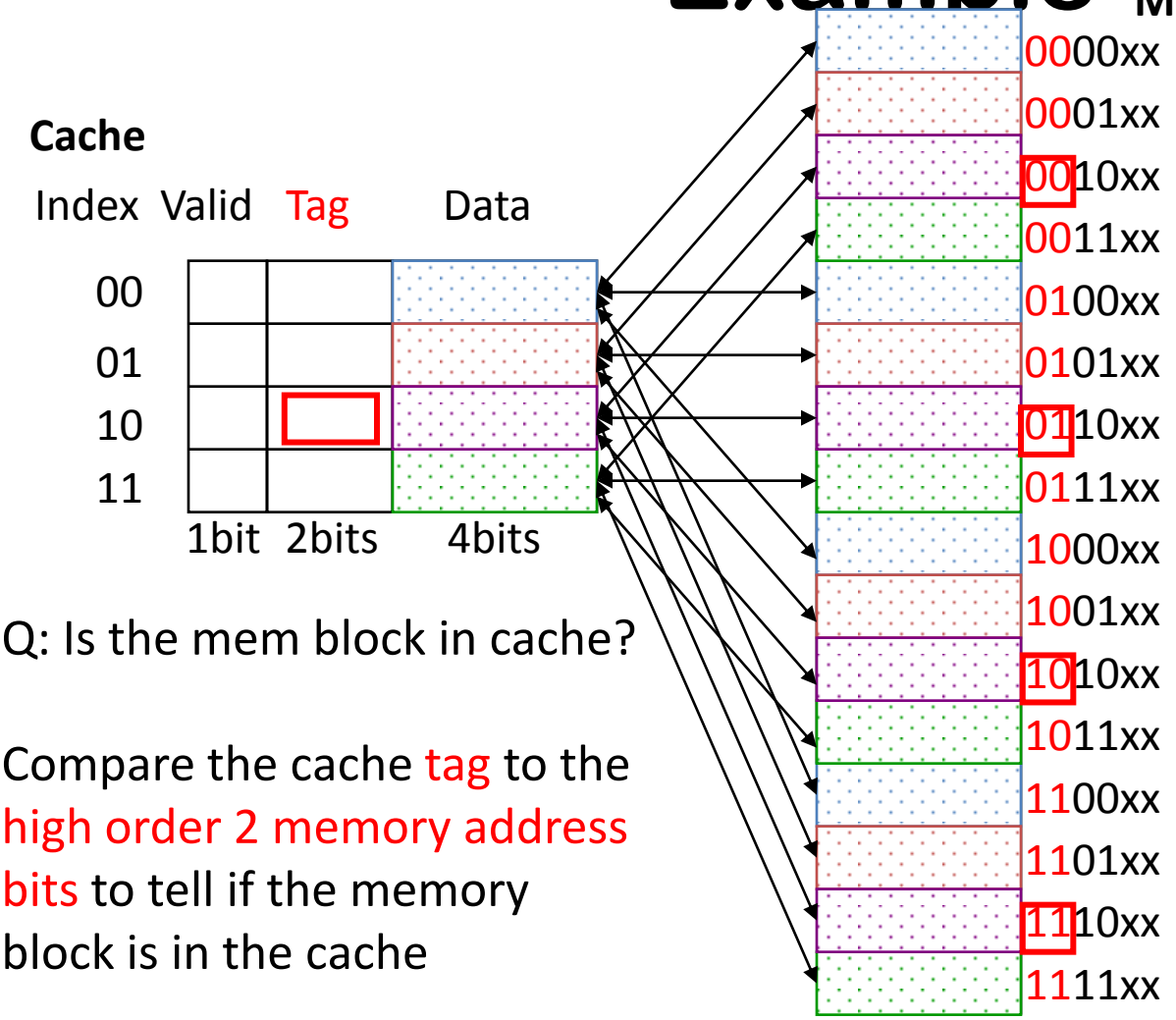
0(0000) 4(0100) 0 4 0 4 0 4



- 8 requests, 8 misses

- Ping pong effect due to conflict misses - two memory locations that map into the same cache block

Recall: Direct Mapped Cache Example



Q: Is the mem block in cache?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

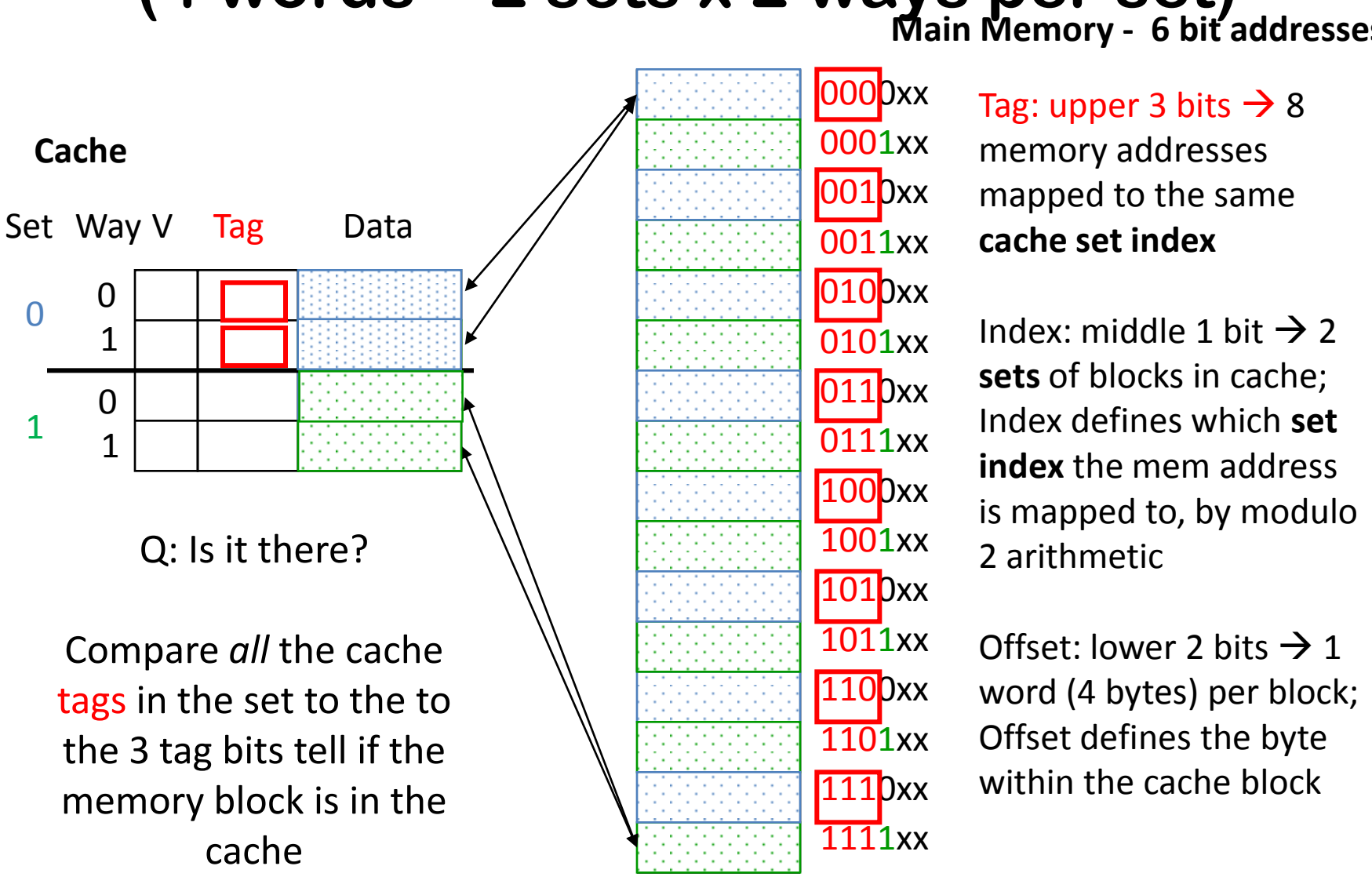
Tag: upper 2 bits → 4 memory addresses mapped to the same cache block index

Index: middle 2 bits → 4 blocks in cache; Index defines which cache block index the mem address is mapped to, by modulo 4 arithmetic

Offset: lower 2 bits → 1 word (4 bytes) per block; Offset defines the byte within the cache block

Index = (block address) modulo (# of blocks in the cache (4))

Example: 2-Way Set Associative $ (4 words = 2 sets x 2 ways per set)



Example: 4-Word 2-Way SA $ Same Reference String

- Consider the sequence of memory accesses

Start with an empty cache - all blocks initially marked as not valid

0(0000) 4(0100) 0 4 0 4 0 4

0 miss

000	Mem(0)

4 miss

000	Mem(0)
010	Mem(4)

0 hit

000	Mem(0)
010	Mem(4)

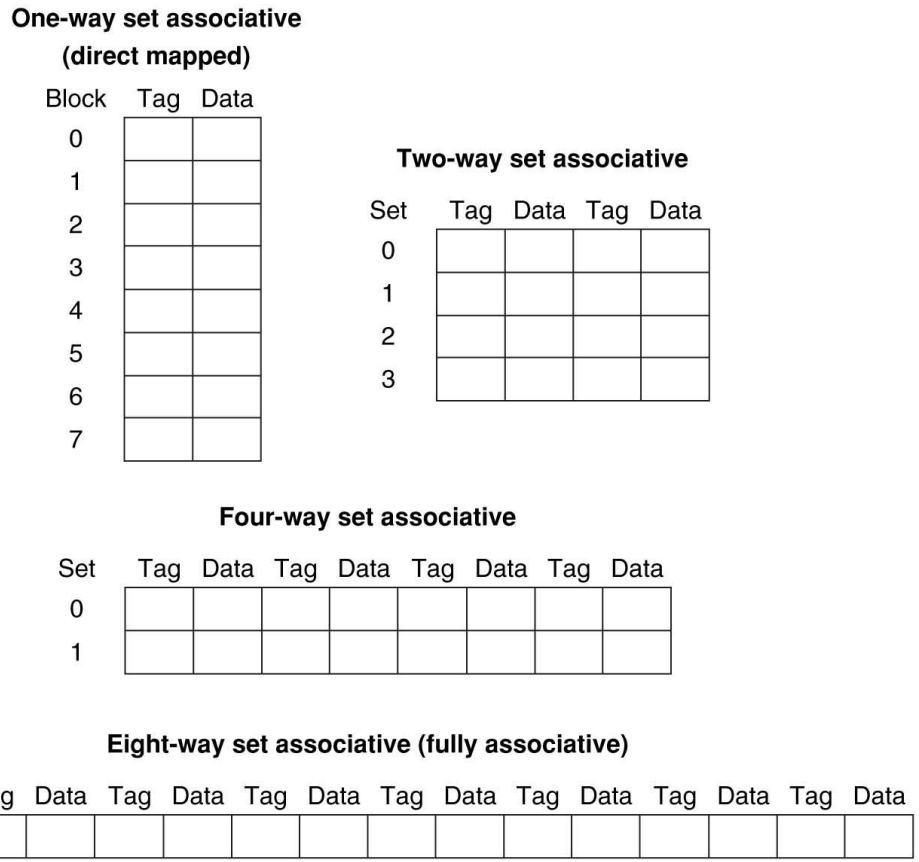
4 hit

000	Mem(0)
010	Mem(4)

- 8 requests, 2 misses

- Solves the ping pong effect in a direct-mapped cache due to conflict misses, since now two memory locations (0 and 4) that map into the same cache set can co-exist!

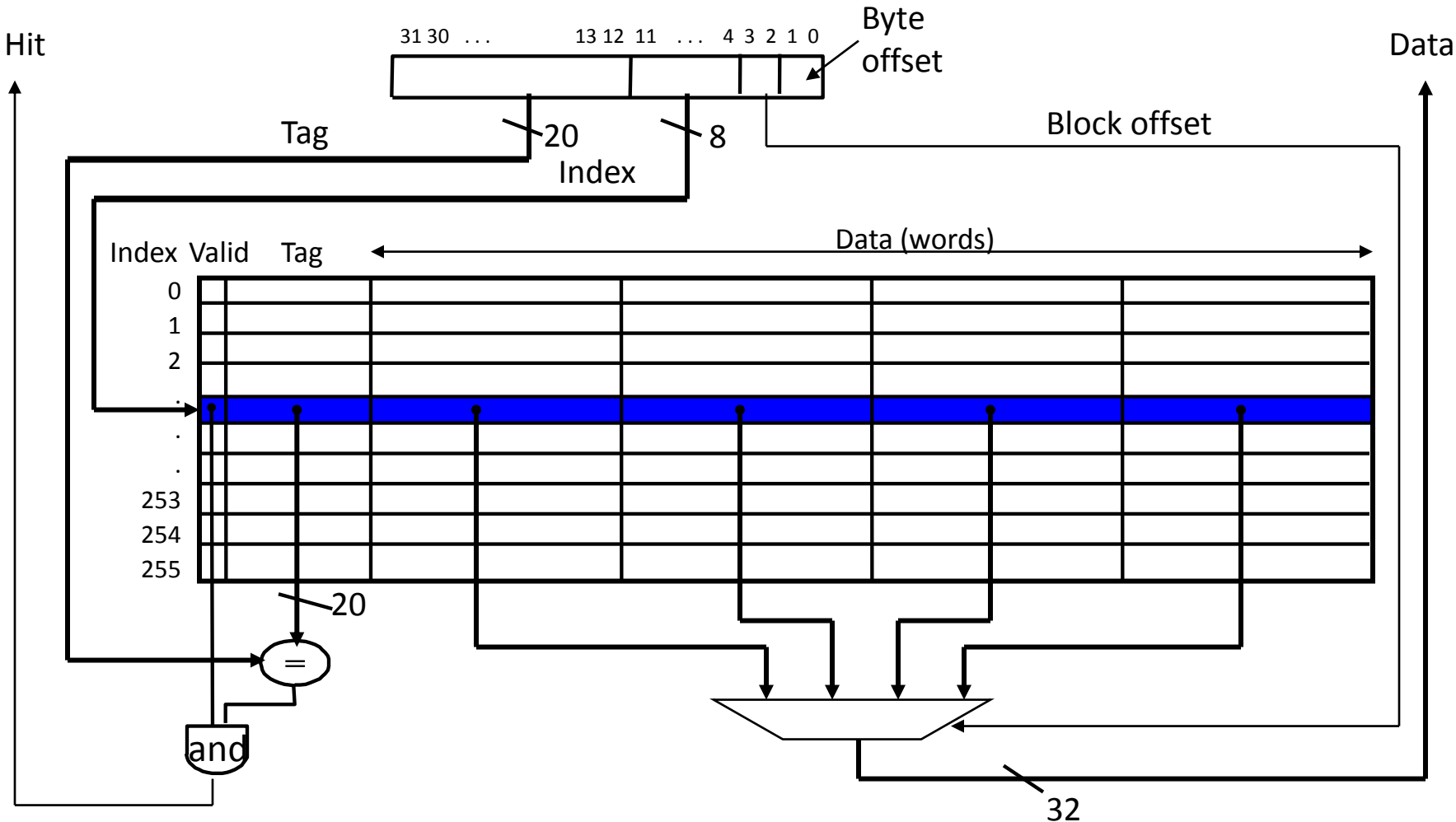
Example: Eight-Block Cache with Different Organizations



Total size of $ in blocks is equal to *number of sets* x *associativity*. For fixed $ size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.

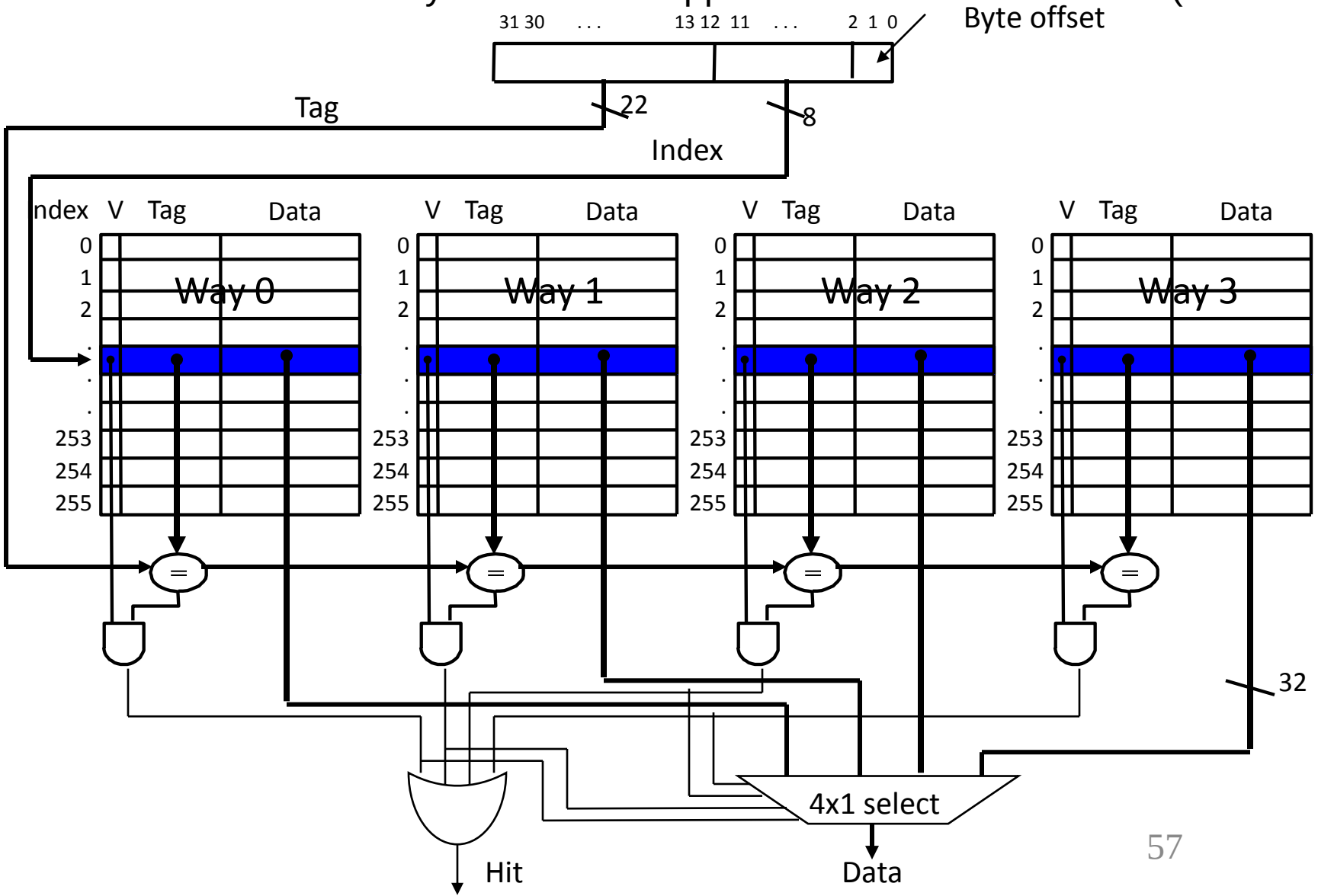
Recall: Direct Mapped Cache Layout Example 2

- 32 bit address space. Tag: 20 bits; Index: 8 bits; Offset: 4 bits
- 16 Bytes/block, cache size = 2^8=256 blocks * 16 Bytes/block = 4K Bytes (1K words)
- 2^20=1 million memory addresses mapped to each cache index (blue line)



Four-Way Set-Associative Cache

- 32 bit address space. Tag: 22 bits; Index: 8 bits; Offset: 2 bits
- 4 Bytes/block, cache size = 4 Ways * 2^8=256 blocks/way * 4 Bytes/block = 4K Bytes (1K words)
- 2^22=4 million memory addresses mapped to each cache set index (blue line)



Explanations

- This is called a 4-way set associative cache because there are 4 cache entries for each cache index. Essentially, you have four direct mapped cache working in parallel.
  - The cache index selects a set from the cache.
  - The four tags in the set are compared in parallel with the upper bits (Tag) of the memory address.
  - If no tags match the memory address tag, we have a cache miss.
  - Otherwise, we have a cache hit and we will select the data from the way where the tag matches occur.

Quiz

- 32 bit address space, 32KB 4-way set associative cache with 8 word blocks. What is the TIO breakdown?

1	T - 21	I - 8	O - 3
2	T - 19	I - 10	O - 3
3	T - 17	I - 12	O - 3
4	T - 19	I - 8	O - 5
5	T - 17	I - 10	O - 5
6	T - 15	I - 12	O - 5

Quzz Answer

- 32 bit address space, 32KB 4-way set associative cache with 8-word blocks. What is the TIO breakdown?

8 word blocks => 32 bytes / block => O = 5
32 KB / (32 bytes / block) = 2^10 blocks total
2^10 blocks / (4 blocks / set) = 2^8 sets total
Index bits will index into sets => I = 8

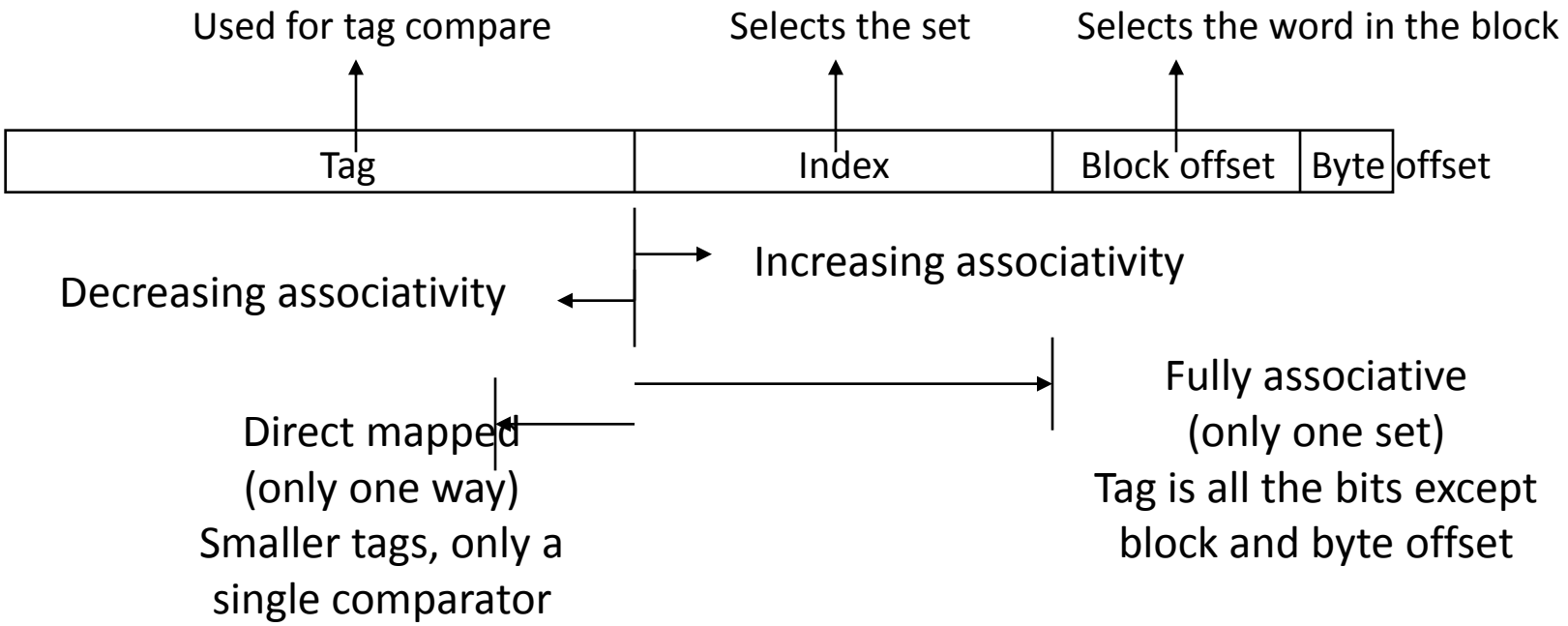
1	T - 21	I - 8	O - 3
2	T - 19	I - 10	O - 3
3	T - 17	I - 12	O - 3
4	T - 19	I - 8	O - 5
5	T - 17	I - 10	O - 5
6	T - 15	I - 12	O - 5

Quiz Answer Cont'

- 32 bit address space, 32KB direct-mapped cache with 8-word blocks.
  - T - 17, I - 10, O – 5
- 32 bit address space, 32KB fully-associative cache with 8-word blocks.
  - T - 27, I - 0, O – 5
- For a direct-mapped cache, when do you have 0 Tag bits?
  - When the cache size is equal to memory size, which is obviously unrealistic. Hence you always have tag bits!

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



Costs of Set-Associative Caches

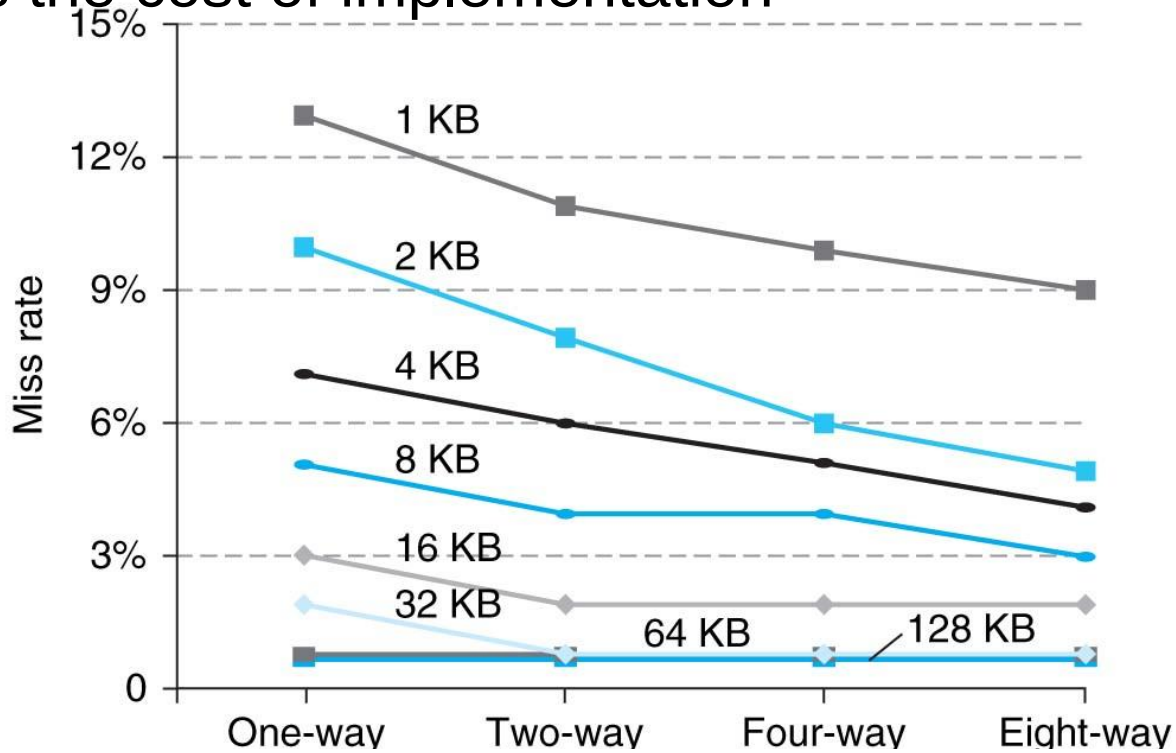
- N-way set-associative cache costs
  - A N-way SA cache will need N parallel comparators for tag comparisons instead of just one comparator for DM cache
  - It is slower than a DM cache because of this extra multiplexer delay. → Hit time is larger.

Cache Block Replacement Policies

- Random Replacement
  - Hardware randomly selects a cache block and throws it out
- Least Recently Used
  - Hardware keeps track of access history
  - Replace the block that has not been used for the longest time

Benefits of Set-Associative Caches

- Choice of DM $ or SA $ depends on the cost of a miss versus the cost of implementation



- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

How to Calculate 3C's using Cache Simulator

1. *Compulsory*: set cache size to infinity and fully associative, and count number of misses
2. *Capacity*: Change cache size from infinity, usually in powers of 2, and count misses for each reduction in size
   - 16 MB, 8 MB, 4 MB, … 128 KB, 64 KB, 16 KB
3. *Conflict*: Change from fully associative to n-way set associative while counting misses
   - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way

Improving Cache Performance: Summary

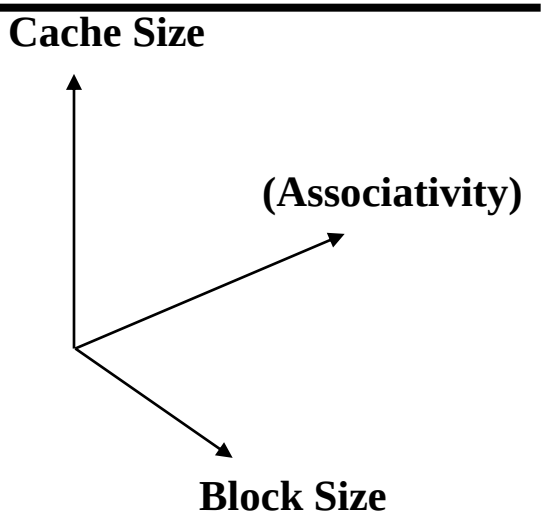
1. Reduce the time to hit in the cache
   - Smaller cache
   - 1 word blocks (no multiplexor/selector to pick word)
2. Reduce the miss rate
   - Bigger cache
   - Larger blocks (16 to 64 bytes typical)
   - More flexible placement by increasing associativity

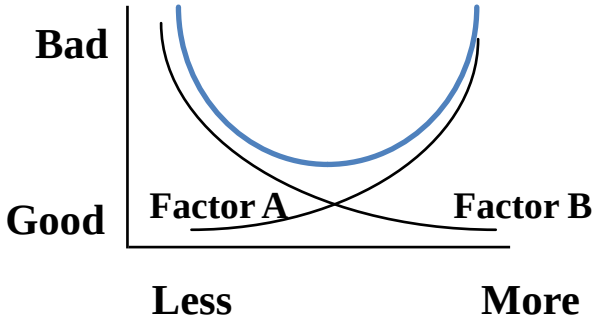
Improving Cache Performance: Summary

3. Reduce the miss penalty
   - Smaller blocks
   - Use multiple cache levels
     - L2 cache not tied to processor clock rate
   - Write-back instead of write-through:
     - Use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading. Check write buffer on read miss – may get lucky
   - Faster backing store/improved memory bandwidth

The Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Write-through vs. write-back
  - Write-allocate vs write-no-allocate
  - Replacement policy (for associative cache)
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins

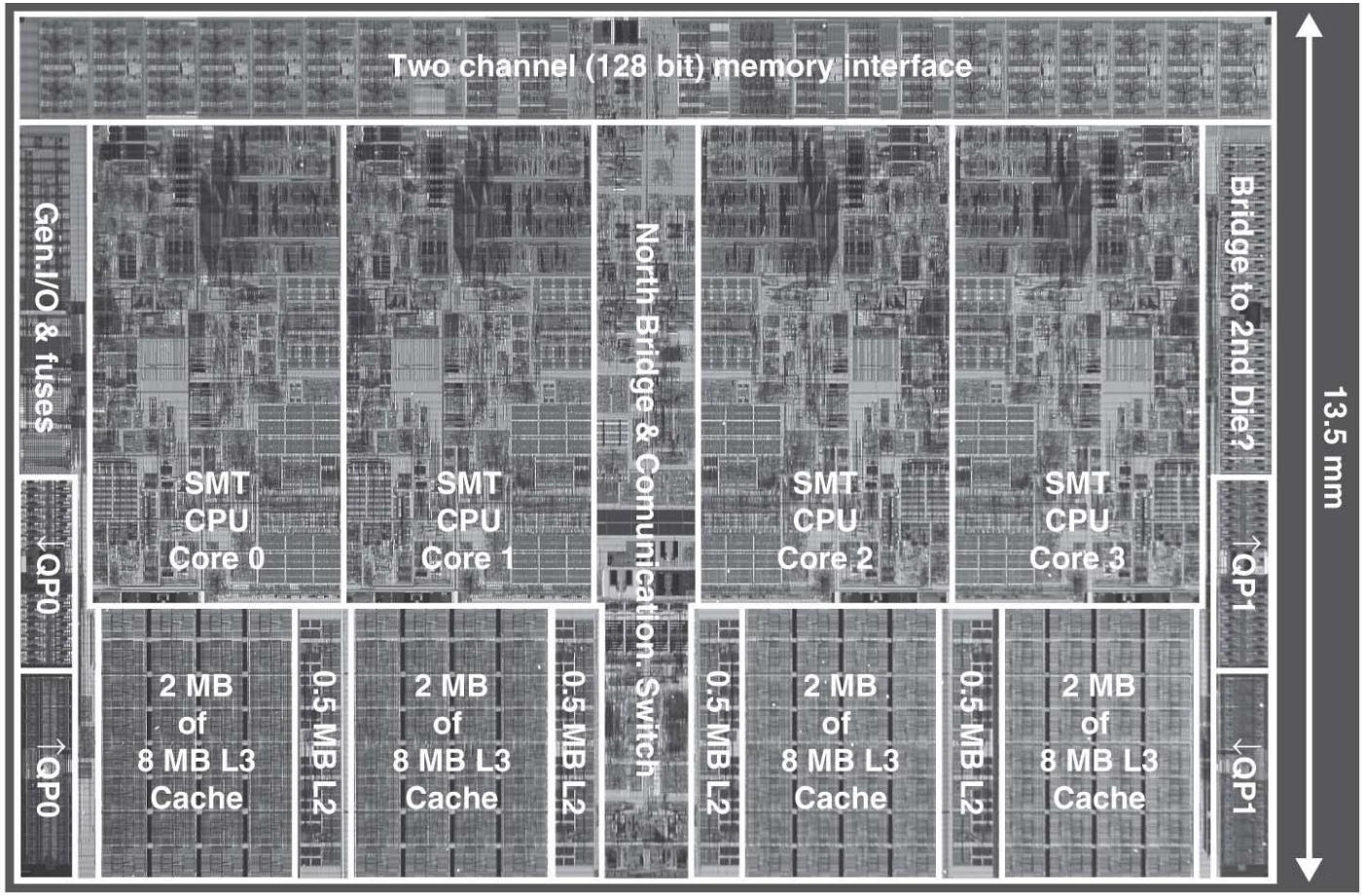




Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 cache associativity	4-way (I), 8-way (D) set associative	2-way set associative
L1 replacement	Approximated LRU replacement	LRU replacement
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 cache associativity	8-way set associative	16-way set associative
L2 replacement	Approximated LRU replacement	Approximated LRU replacement
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 cache associativity	16-way set associative	32-way set associative
L3 replacement	Not Available	Evict block shared by fewest cores
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

- L1→L2→L3
  - Size increasing
  - Associativity increasing
  - Both cause hit time to increase, and miss rate to decrease

Intel Nehalem Die Photo



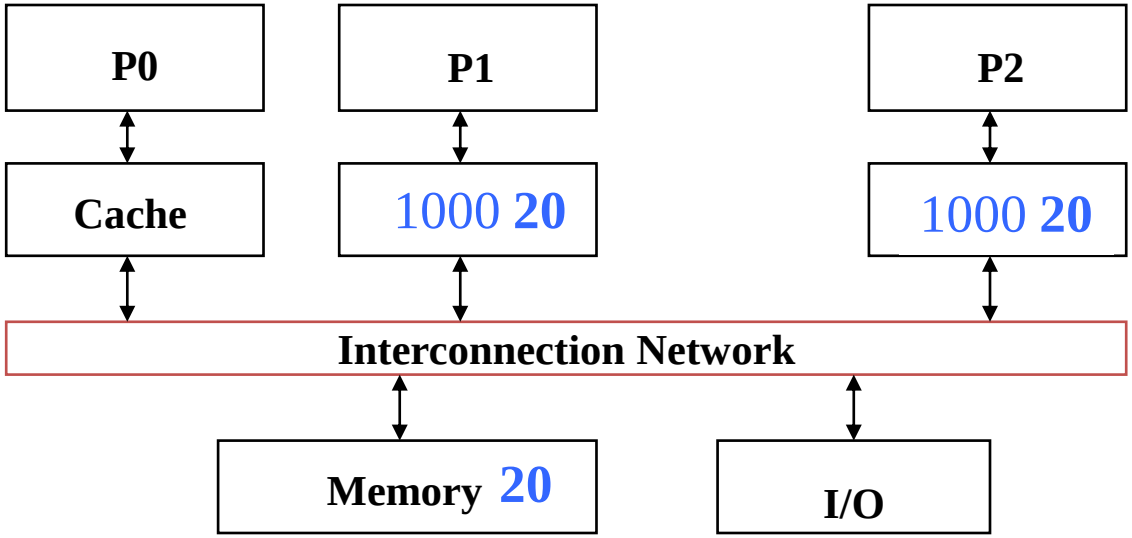
- 4 cores, 32KB I$/32-KB D$ L1$, 512KB L2$. L1$ and L2$ are private to each core
- Share one single 8-MB L3$

Keeping Multiple Caches Coherent

- HW Architect's job: shared memory => keep cache values coherent
- One approach: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies, and write through to memory
- Shared written result can "ping-pong" between caches

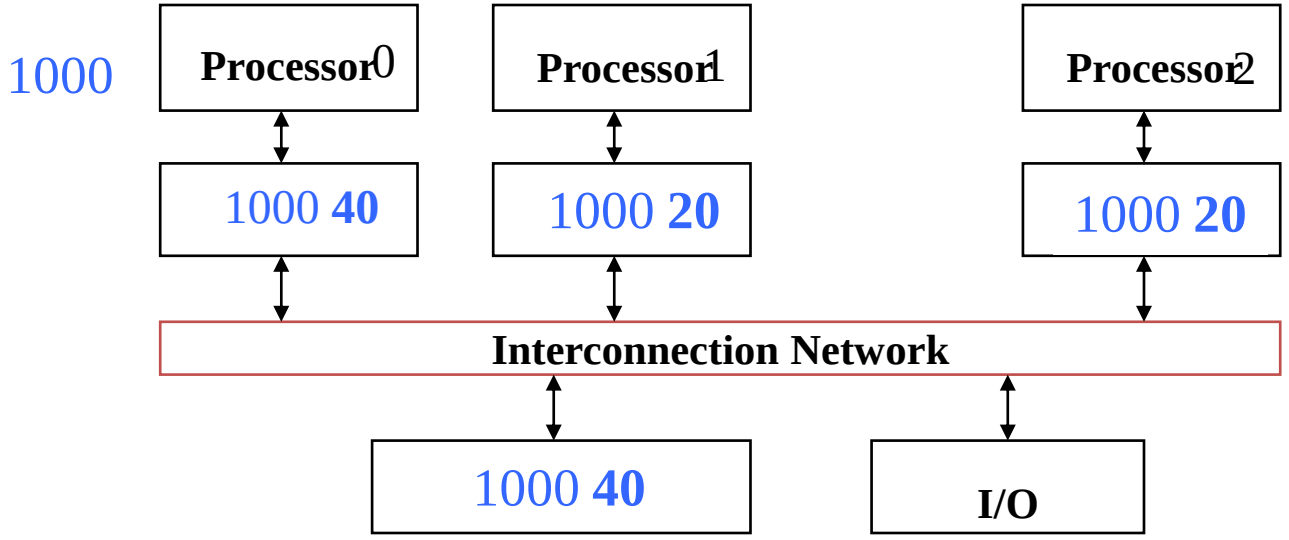
Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000] (value 20)



Shared Memory and Caches

- What if?
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Processor 0
Write
Invalidates
Other Copies

False Sharing

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose X is at address 4000, Y in 4012
  - Two variables are in the same cache block, even though they have different addresses
- Effect called *false sharing*

Summary #1/2: Cache

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size

Summary #1/2: Cache

- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent
- Set-associativity - Reduce Cache Miss Rate
  - Memory block maps into more than 1 cache block
  - N-way: N possible places in cache to hold a memory block
- Multi-level caches - Reduce Cache Miss Penalty
  - Optimize first level to be fast!
  - Optimize 2nd and 3rd levels to minimize the memory access penalty
- Lots of cache parameters!
  - Write-back vs. write through, write-allocate, block size, cache size, associativity, etc.

Quiz I

- Q: Consider 32-bit address space; a direct-mapped cache with size 16KB;each cache block is 4 words. What is the TIO breakdown?
- A:
- Cache size = 16KB = 16 * 2^10 bytes
- Cache block size = 4 words = 4 * 4 bytes = 16 bytes = 2^4
- Number of cache blocks= 16 * 2^10 bytes / 16 bytes = 2^10
- Index bits = 10
- Offset bits = 4
- Tag bits = 32 – 10 – 4 = 18

Quiz II

- Q: Consider 32-bit address space; a two-way set-associative cache with size 16KB;each cache block is 4 words. What is the TIO breakdown?
- A:
- Cache size = 16 * 2^10 bytes
- cache block size = 16 bytes
- Set size = cache block size * set associativity = 16 bytes * 2 = 32 bytes
- Number of sets = 16 * 2^10 bytes / 32 bytes = 2^9
- Index bits = 9
- Offset bits = 4
- Tag bits = 32 – 9 – 4 = 19

Quiz III

- Q: How many 32-bit integers can be stored in a byte-addressed direct-mapped cache with 15 tag bits, 15 index bits, and 2 offset bits?
- A: Each cache block is 2^2=4 Bytes and can store one 32-bit integer. The cache has a total number of 2^15=32K blocks, hence it can store 32K integers.

Quiz IV

- Consider 8-bit address space; a direct mapped, write-back, write-allocate cache that can hold two blocks of 8 Bytes each. The cache is initially empty. The following sequence of memory operations are made, where each reference is a byte address of a 4-byte number (Only consider word aligned word addresses, i.e. locations 0, 4, 8, and so on. lw: load word; sw: store word) :
  - lw 0
  - sw 44
  - lw 52
  - lw 88
  - lw 0
  - sw 52
  - lw 68
  - lw 44
- Work out the cache behavior after each operation.

Quiz IV Answer

- Tag: 4 Index: 1 Offset: 3. The low-order 3 bits of an address specifies the byte address, and the next 1 bit is the index. I'll write addresses as a triple of tag:index:offset. We have:
- lw 0 = 0000:0:000
  - Bytes 0-7 loaded into cache index 0.
- sw 44= 0010:1:100
  - Bytes 40-47 loaded into cache index 1; bytes 44-47 modified; block marked "dirty".
- lw 52 = 0011:0:100
  - Bytes 48-55 loaded into cache index 0; Clean miss (since replaced block was clean), previous block discarded
- lw 88 = 0101:1:000
  - Bytes 88-95 loaded into cache index 1; Dirty miss (since replaced block was dirty); previous (dirty) contents written back to memory; block marked "clean“
- lw 0 = 0000:0:000
  - Bytes 0-7 loaded into cache index 0. Clean miss; block marked “clean”
- sw 52 = 0011:0:100
  - Bytes 48-55 loaded into cache index 0. Clean miss. bytes 52-55 modified; block marked "dirty".
- lw 68 = 0010:0:100
  - Bytes 64-71 brought into cache index 0; Dirty miss; previous (dirty) contents written back to memory; block marked “clean”.
- lw 44 = 0010:1:100
  - Bytes 40-47 loaded into cache index1; Clean miss; block marked "clean".

Quiz V

A. For a given cache size: a larger block size can cause a lower hit rate than a smaller one.

B. If you know your computer's cache size, you can often make your code run faster.

C. Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

	ABC
1:	FFF
1:	FFT
2:	FTF
2:	FTT
3:	TFF
3:	TFT
4:	TTF
5:	TTT

Quiz V Answer

A. Yes – if the block size gets too big, fetches become more expensive and the big blocks force out more useful data.

B. Certainly! That's call "tuning"

C. "Most Recent" items ⇒ Temporal locality

	ABC
1:	FFF
1:	FFT
2:	FTF
2:	FTT
3:	TFF
3:	TFT
4:	TTF
5:	TTT