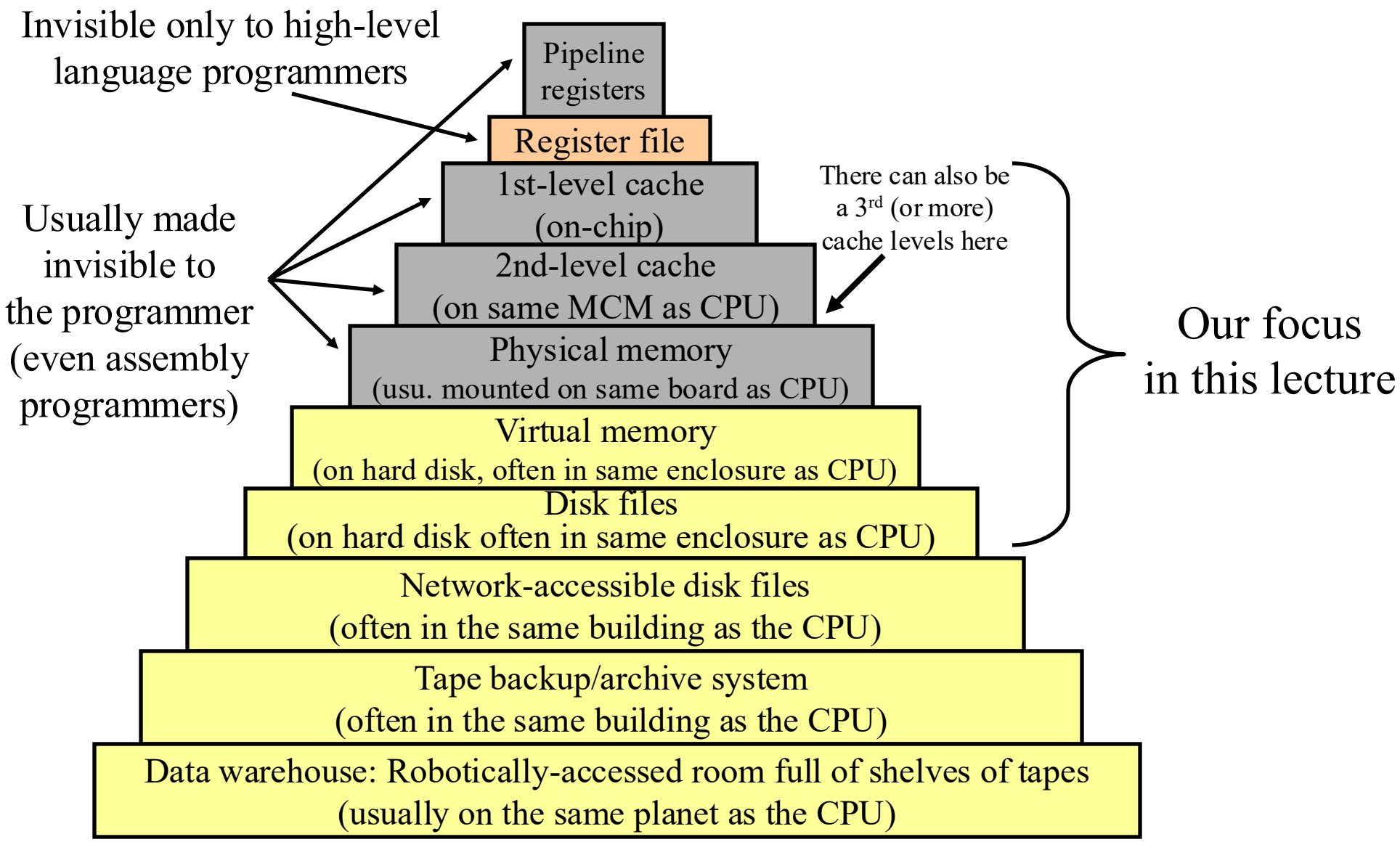
Review of Memory Hierarchy

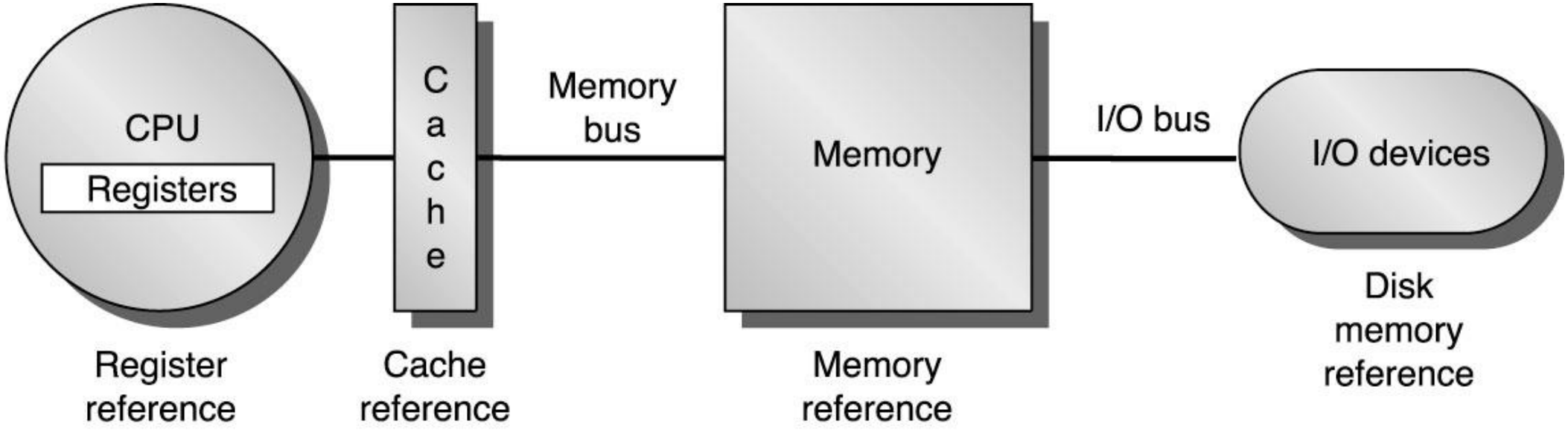
- Introduction
- Cache Basics
- Cache Performance
- Six Basic Cache Optimizations
- Virtual Memory
- Conclusion

Many Levels in Memory Hierarchy



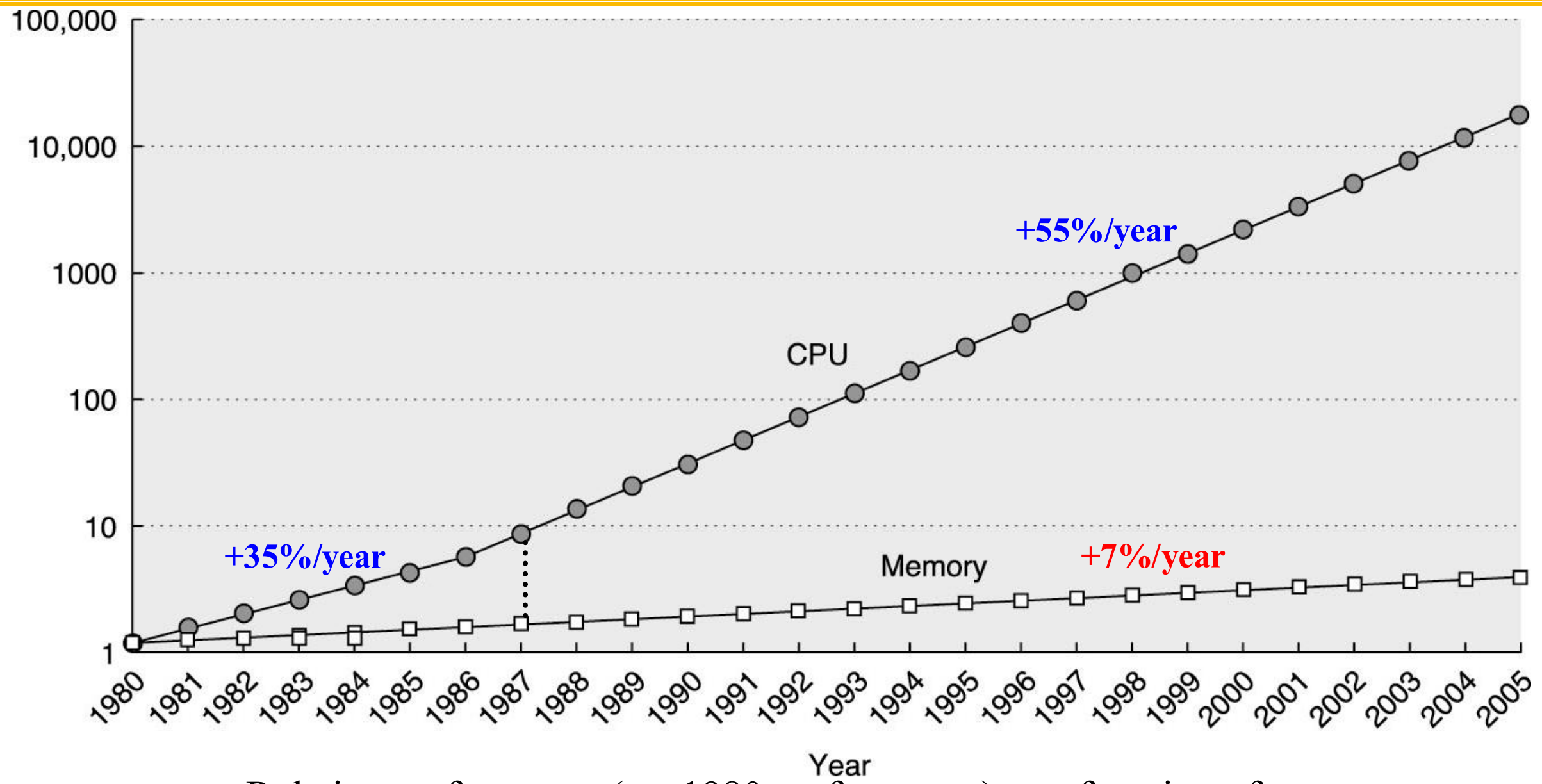
Simple Hierarchy Example

- Orders of magnitude change in characteristics between levels:



Size	**<4 KiB**	**32KiB – 8 MiB**	**<1TB**	**>1 TB**
Access Time (ns)	**0.1-0.2**	**0.5-10**	**30-150**	**5,000,000**
Bandwidth (MiB/s)	**1,000,000 – 10,000,000**	**20,000-50,000**	**10,000-30,000**	**100-1000**
Managed by	**Compiler**	**Hardware**	**Operating System**	**Operating System**

CPU vs. Memory Performance Trends



Relative performance (vs. 1980 performance) as a function of year

Review of Memory Hierarchy

- Introduction
- Cache Basics
- Cache Performance
- Six Basic Cache Optimizations
- Virtual Memory
- Conclusion

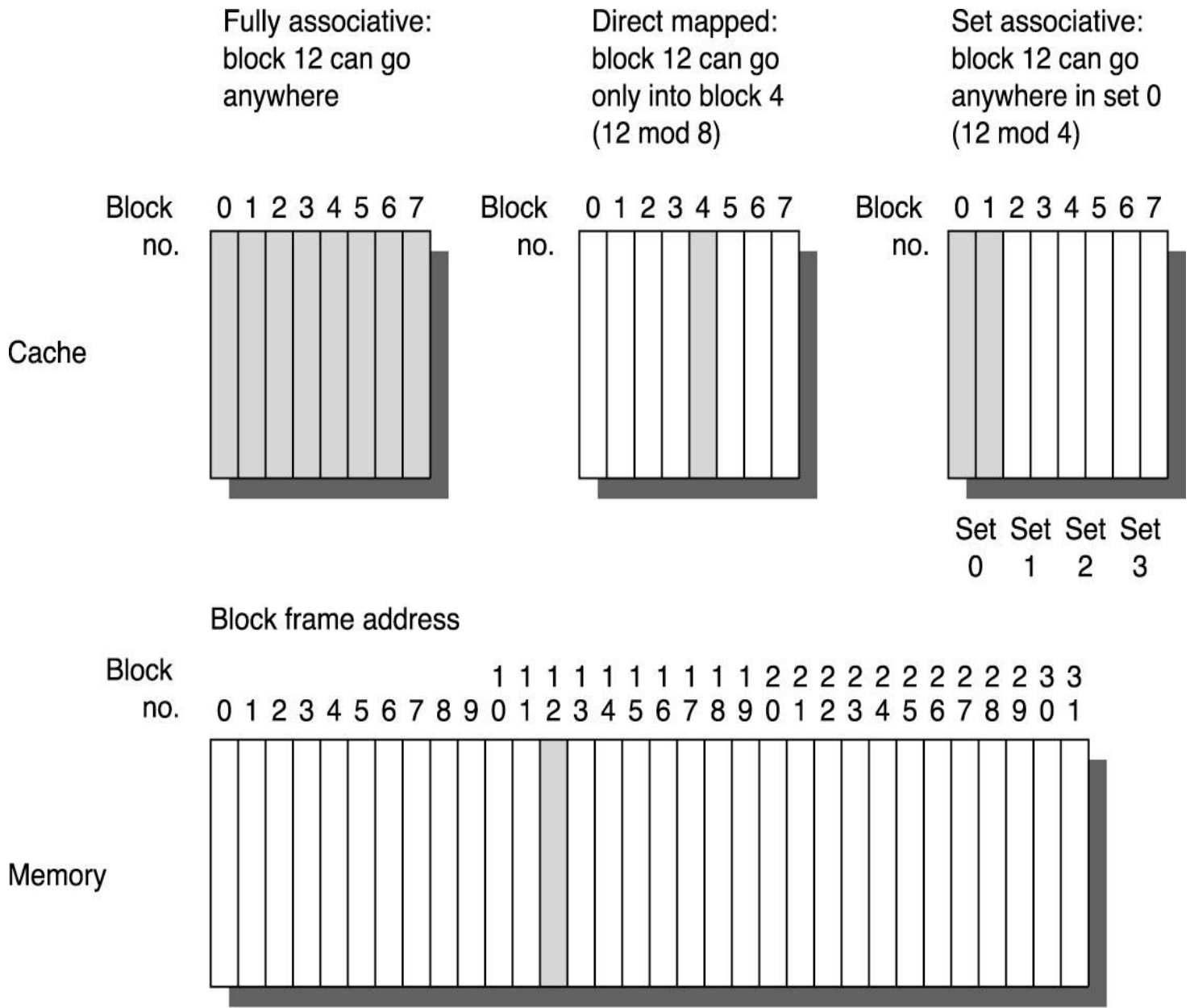
Principle of Locality

- Programs access a small portion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - For example, instructions in a loop, induction variables
- Spatial locality
  - Items near the accessed now are likely to be accessed soon
  - For example, sequential instruction access, array data

Four Basic Questions

- Consider levels in a memory hierarchy.
  - Use block as unit of data transfer between cache levels and memory; satisfy Principle of Locality.
- The level design is described by four behaviors
  - Block Placement:
    - Where could a new block be placed in the level?
  - Block Identification:
    - How is a block found if it is in the level?
  - Block Replacement:
    - Which existing block should be replaced if necessary?
  - Write Strategy:
    - How are writes to the block handled?

Block Placement Schemes



Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Three Cache Choices

- We will use the following example to understand the difference of three cache choices:
  - Direct Cache
  - Fully-Associative Cache
  - Set-Associative Cache
- lw x1, 48 (x0) will lead to address of 48 (i.e., x0 + 48) =

  0000 0000 0000 0000 0000 0000 0011 0000
- Assume the word (block) size of 32 bits (4 bytes). Therefore, the word (block) offset would be **2** bits ("00" shown in red).
- Consider the next 4 bits ("1100") as the block number to compare the three cache choices.

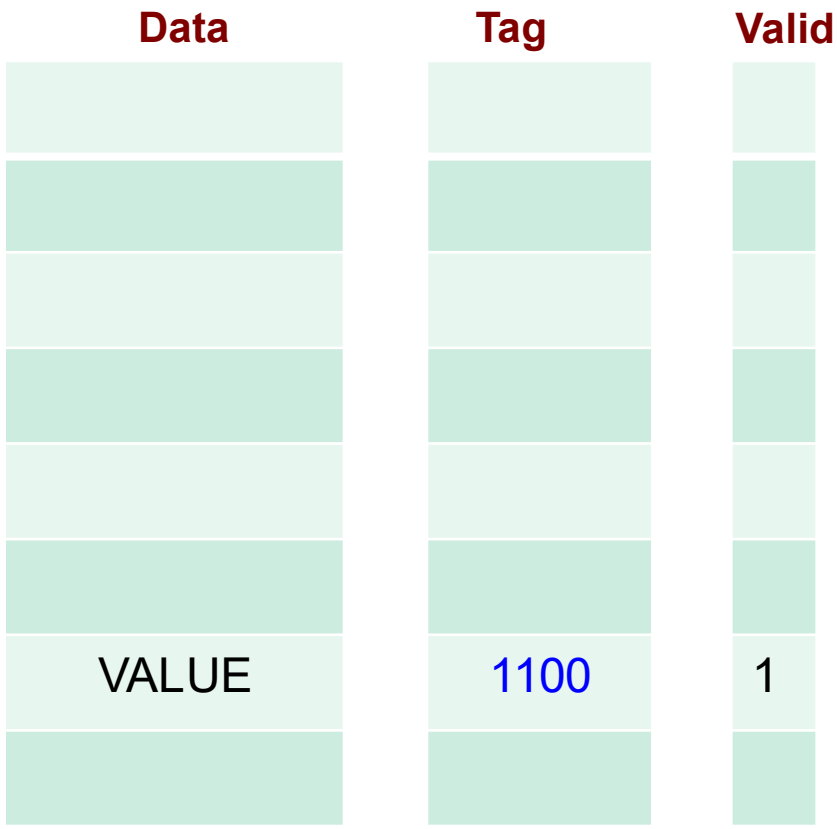
Direct Cache: Example

- Now consider the following eight entry directed cache (implicit index of 3 bits, "100" out of "1100").
- Therefore, we have to store the leftmost "1" from "1100" as the tag.
  - In reality, 32-3-2 bits are tag ➔ 0000 0000 0000 0000 0000 0000 001

Implicit Address	Data	Tag	Valid
000			
001			
010			
011			
100	VALUE	1	1
101			
110			
111			

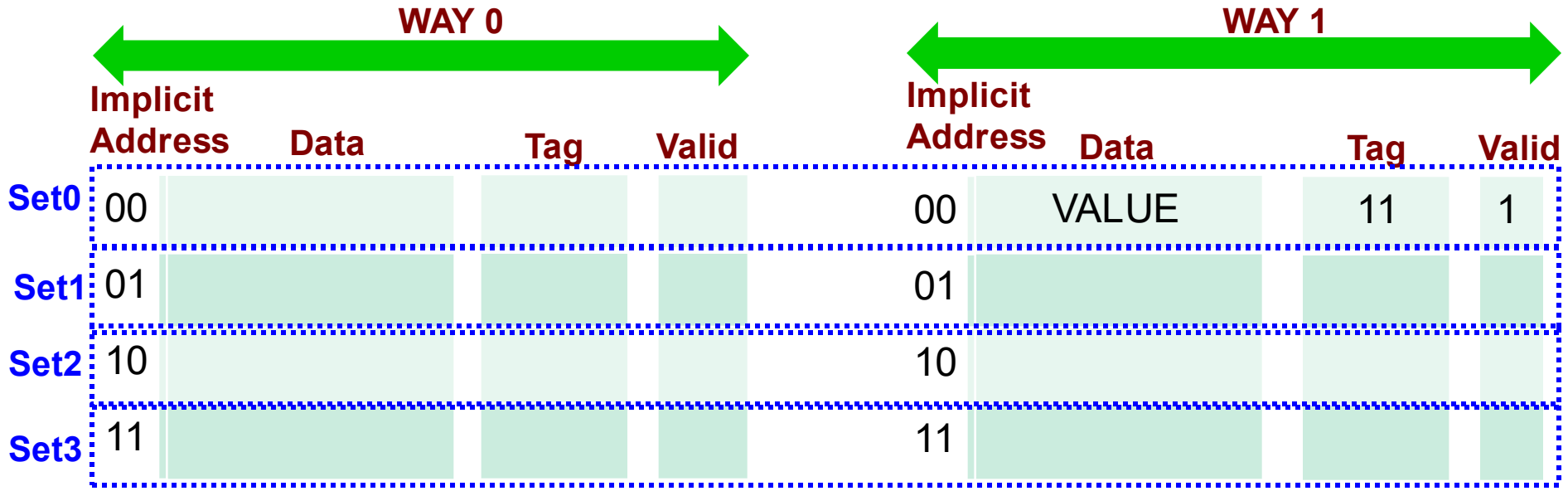
Fully-Associative Cache: Example

- Now consider the following eight entry fully-associative cache The data for block address “1100” can go anywhere, let us randomly place it in the sixth position.
- Therefore, we have to store “1100” as the tag.
  - In reality, 32-**2** bits are tag ➔ 0000 0000 0000 0000 0000 0000 001100

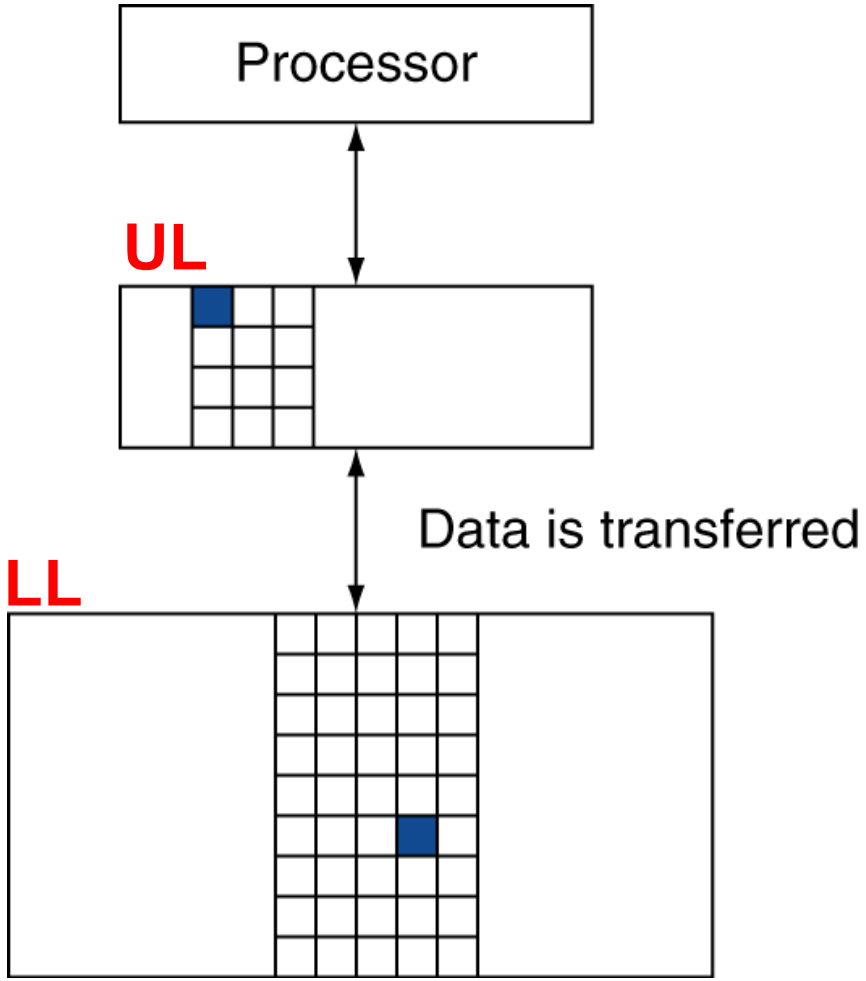


Set-Associative Cache: Example

- Now we create a 2-way set-associate cache from 8 blocks (4 blocks each way ➔implicit index of 2 bits, “00” out of “1100”).
- We have to store “11” from “1100” as the tag.
  - In reality, 32-2-2 bits are tag ➔ 0000 0000 0000 0000 0000 0000 0011
- The data can go to any of the two choices in Set 0. Let us randomly place it in the right one (Way 1).



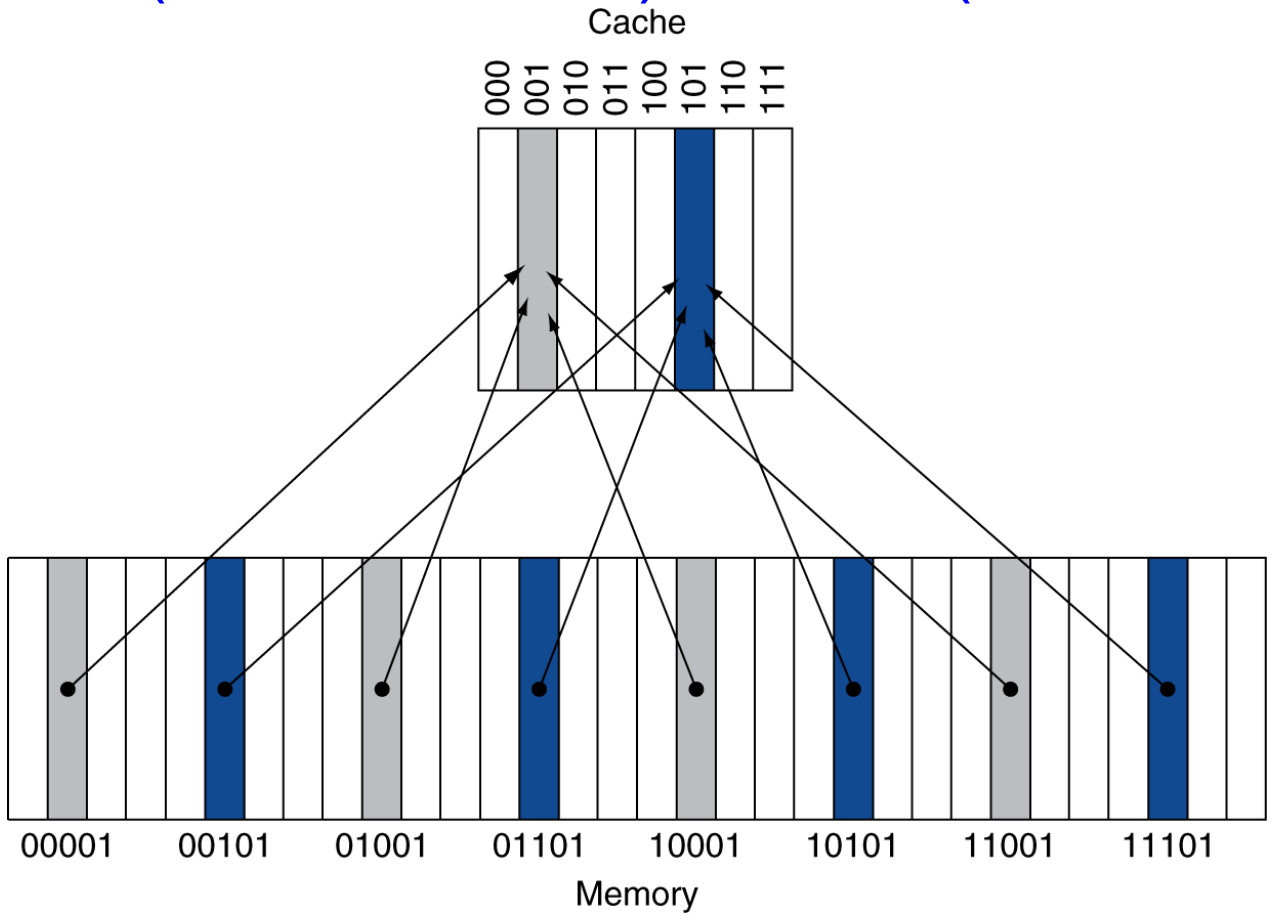
Memory Hierarchy Levels



- Block (a.k.a. Line): unit of copying
  - May be multiple words
- If accessed data is present in upper level (UL)
  - Hit
  - Hit ratio = $\frac{Number\ of\ hits}{Number\ of\ accesses}$
- If accessed data is absent in upper level (UL)
  - Miss: block copied from the lower level (LL)
    - Time taken: miss penalty
    - Miss ratio = $\frac{Number\ of\ misses}{Number\ of\ accesses}$ = 1 – hit ratio
  - The data is supplied from the upper level

Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)



- **#Blocks is a power of 2**
- **Use low-order address bits**

Example of Cache Accesses

- Empty load/store (LD/ST) buffer

Word addr	Binary addr	Hit/miss	Cache block

- 8-blocks, 1 word/block, direct mapped
- Initial state of L1 cache

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

L1 Cache

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Fetch from L2

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110

L1 Cache

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010

L1 Cache

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Fetch from L2

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010
16	10 000	Miss	000

L1 Cache

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Fetch from L2

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010
16	10 000	Miss	000
3	00 011	Miss	011

L1 Cache

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Fetch from L2

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

L1 Cache

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000
18	10 010	Miss	010

L1 Cache

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Replace to L2

Example of Cache Accesses

LD/ST Buffer (Processor)

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110
22	10 110	Hit	110
26	11 010	Miss	010
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000
18	10 010	Miss	010

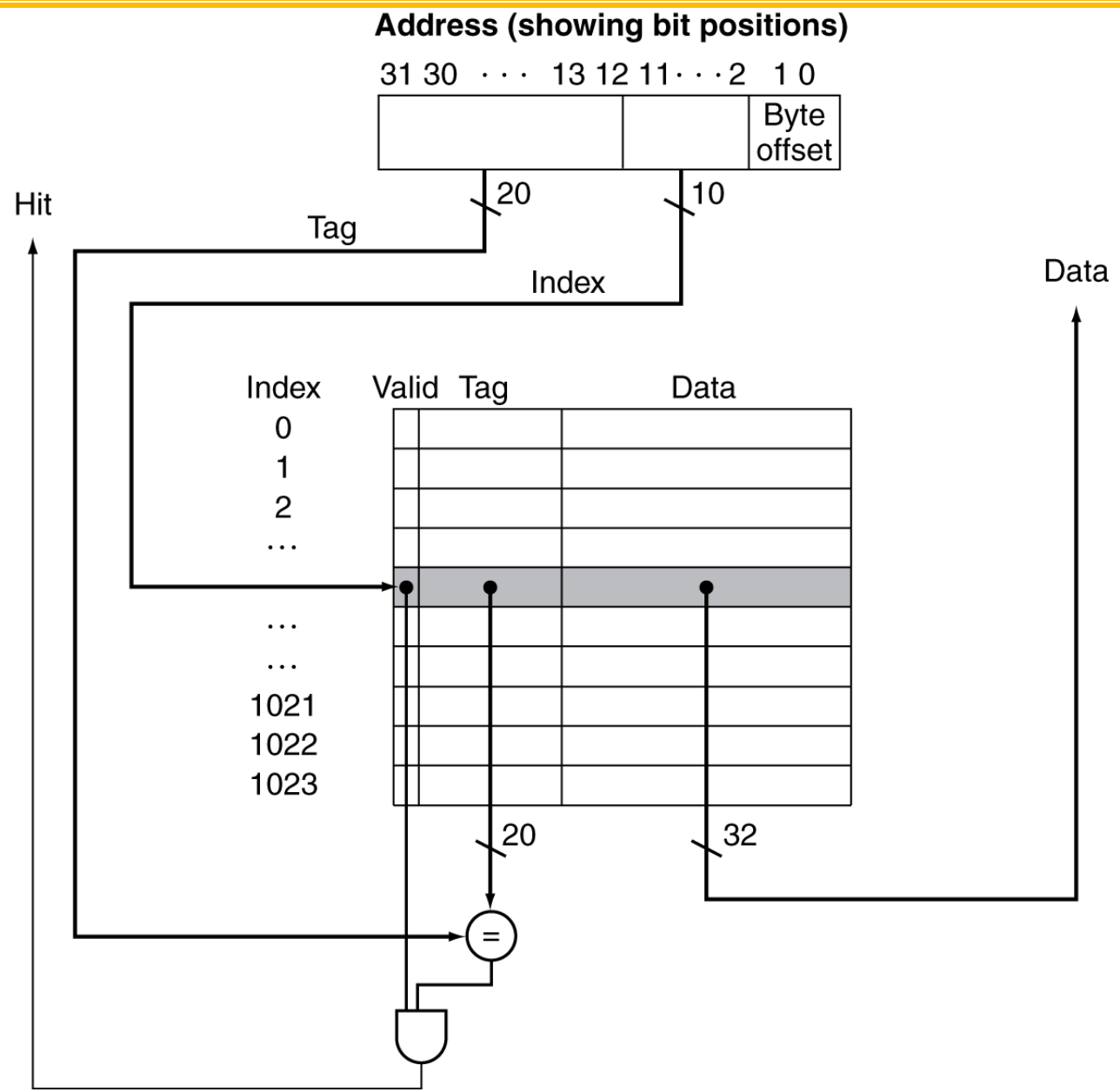
Hit Ratio = $\frac{2}{7}$ = 28.57%

L1 Cache

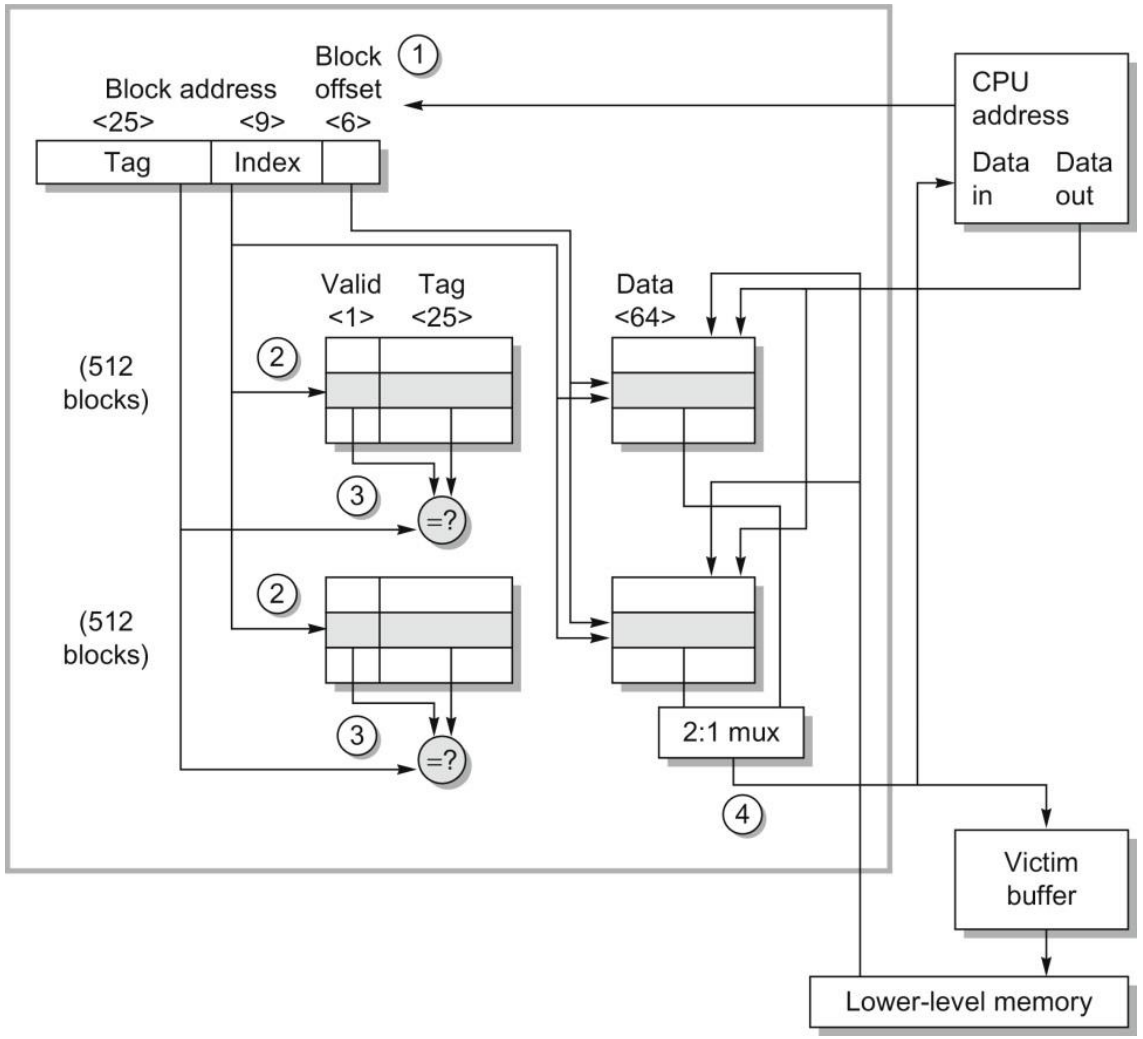
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Fetch from L2

Address Subdivision Implementation



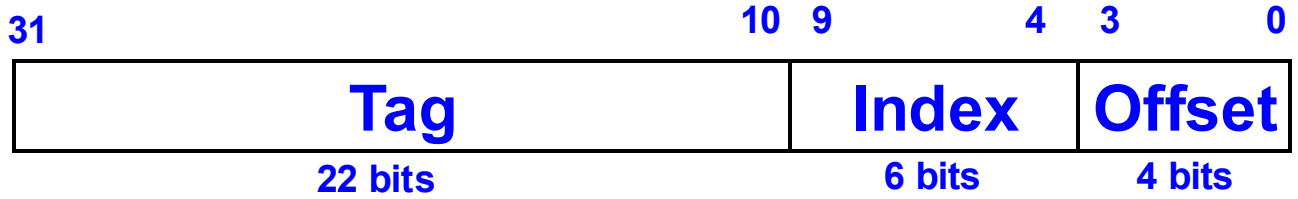
Data Cache in Opteron



64 KiB cache is 2-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes.

Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Naïve method



$(1200)_{10}$ = (00000000000000000001 001011 0000)$_2$

- Alternative Method
  - Block address = $\lfloor 1200/16 \rfloor$ = 75
  - Block number = 75 modulo 64 = 11

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Write-Through

- On data-write hit, could just update the block in cache
  - Then cache and memory would be inconsistent
- Write-through: also update memory
- Write-through policy makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, writes to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

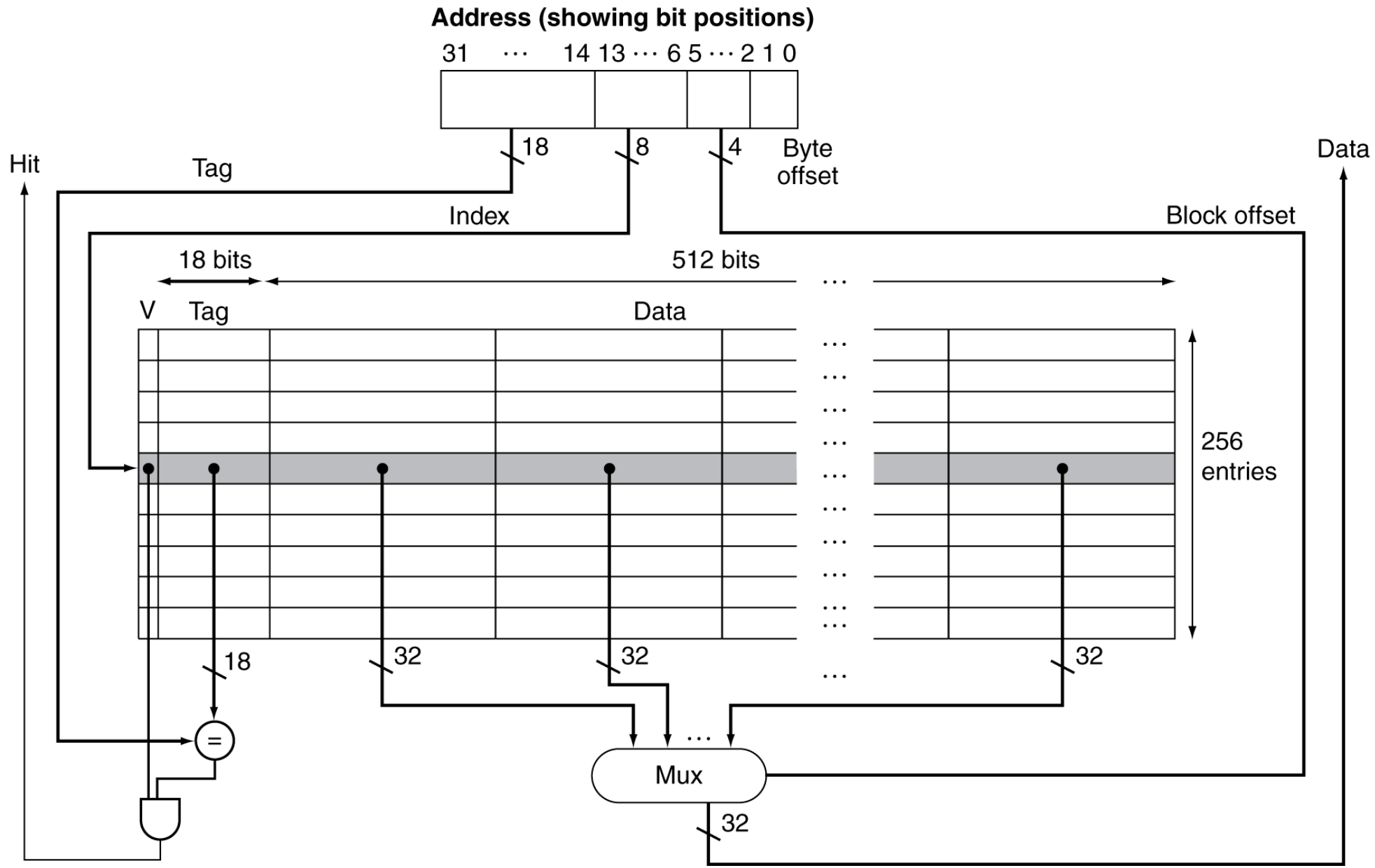
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Fetch the block on write miss.
    - If write-allocate is not followed, it can lead to data inconsistency

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

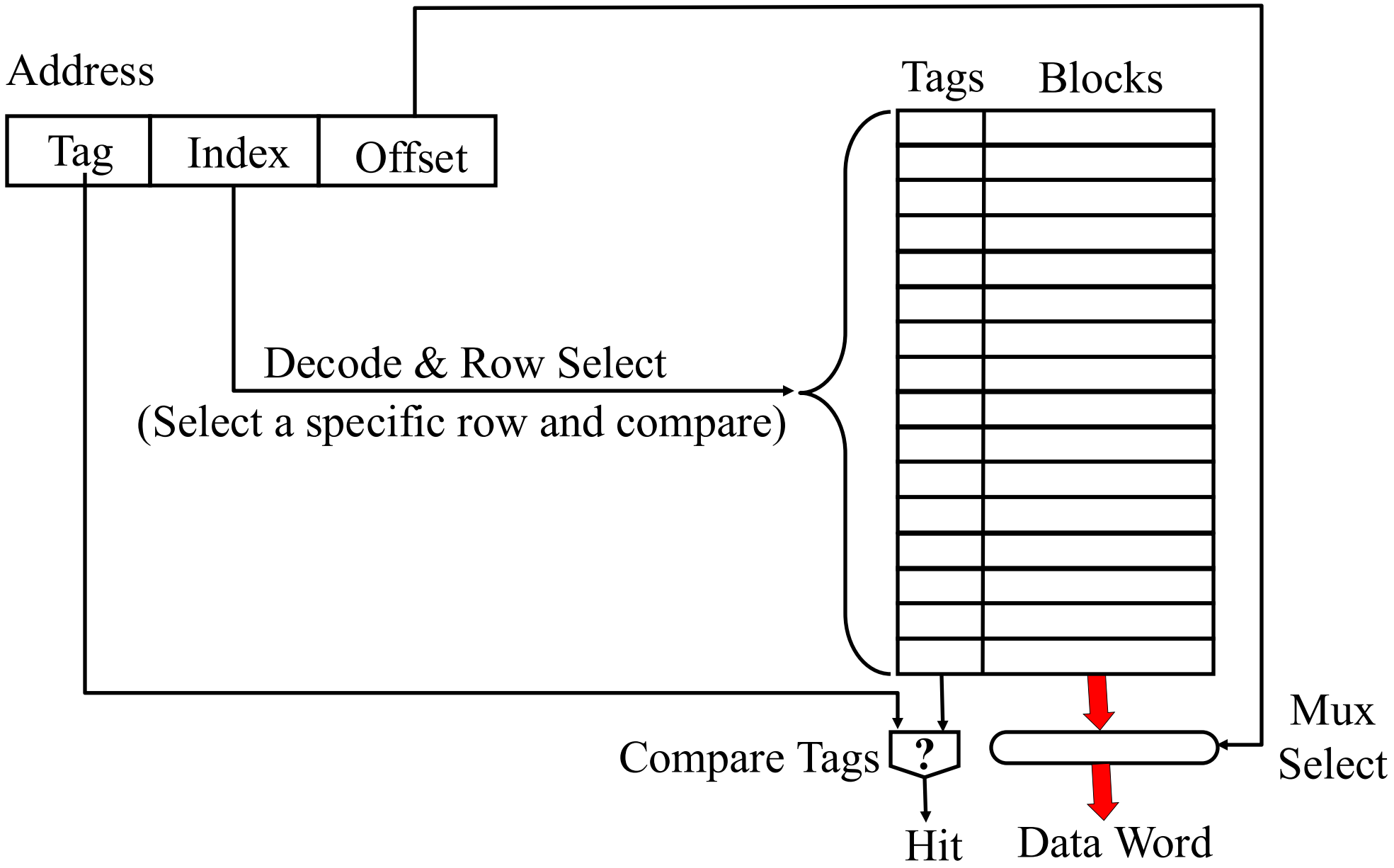
Example: Intrinsity FastMATH



Direct-Mapped Placement

- A block can only go into one frame in the cache
  - Determined by block's address (in memory space)
    - Frame number usually given by some low-order bits of block address.
- This can also be expressed as:
  - (Frame number) = (Block address) ***mod*** (Number of frames/sets in cache)
- In a direct-mapped cache
  - block placement & replacement are both completely determined by the address of the new block that is to be accessed.

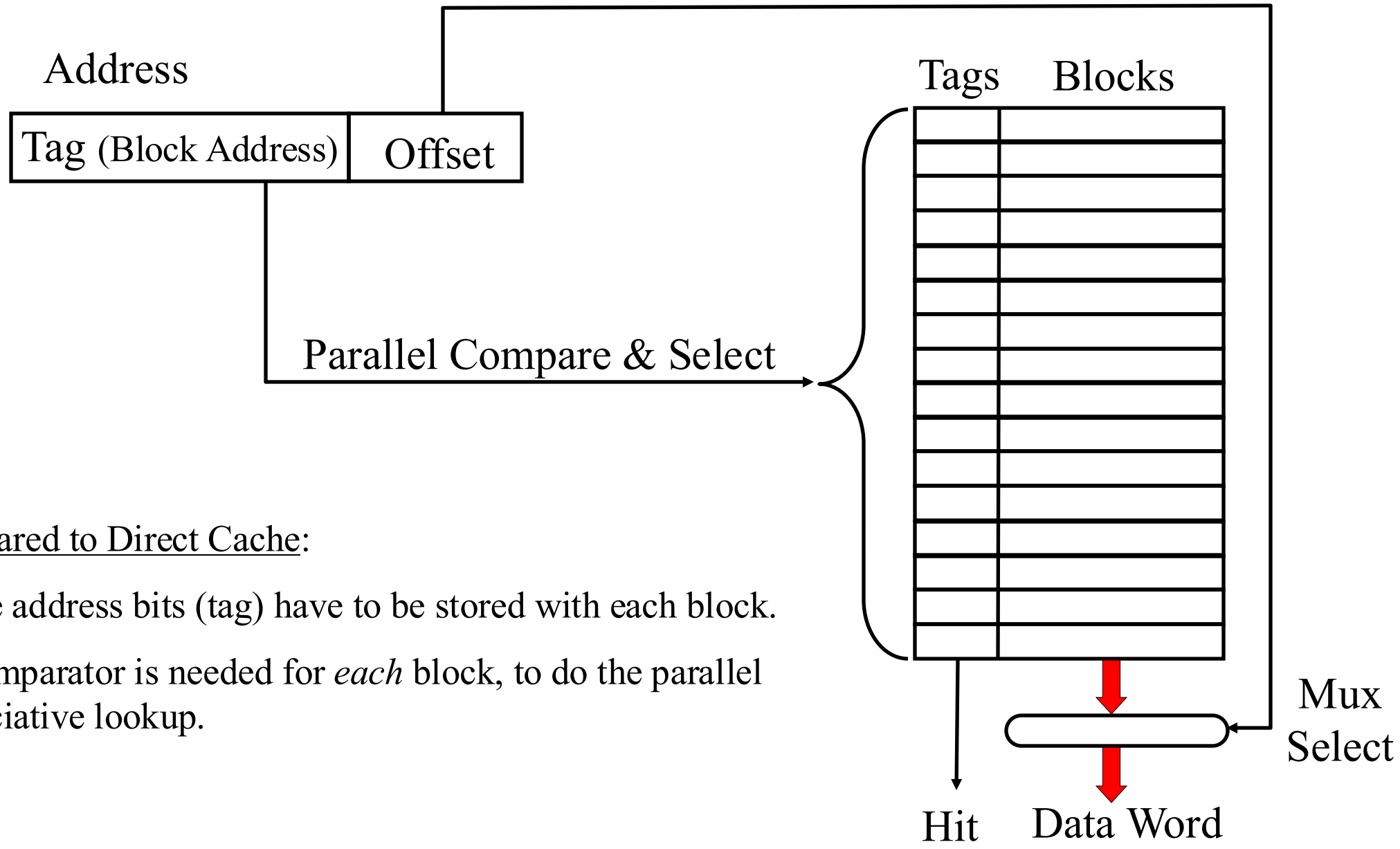
Direct-Mapped Identification



Fully-Associative Placement

- One alternative to direct-mapped is:
  - Allow block to fill ***any*** empty frame in the cache.
- How do we then locate the block later?
  - Can *associate* each stored block with a *tag*
    - Identifies the block's location in cache.
  - When the block is needed, treat the cache as an *associative memory*, using the tag to match all frames in parallel, to pull out the appropriate block.
- Another alternative to direct-mapped is placement under full program control.
  - A register file can be viewed as a small programmer-controlled cache (w. 1-word blocks).

Fully-Associative Identification



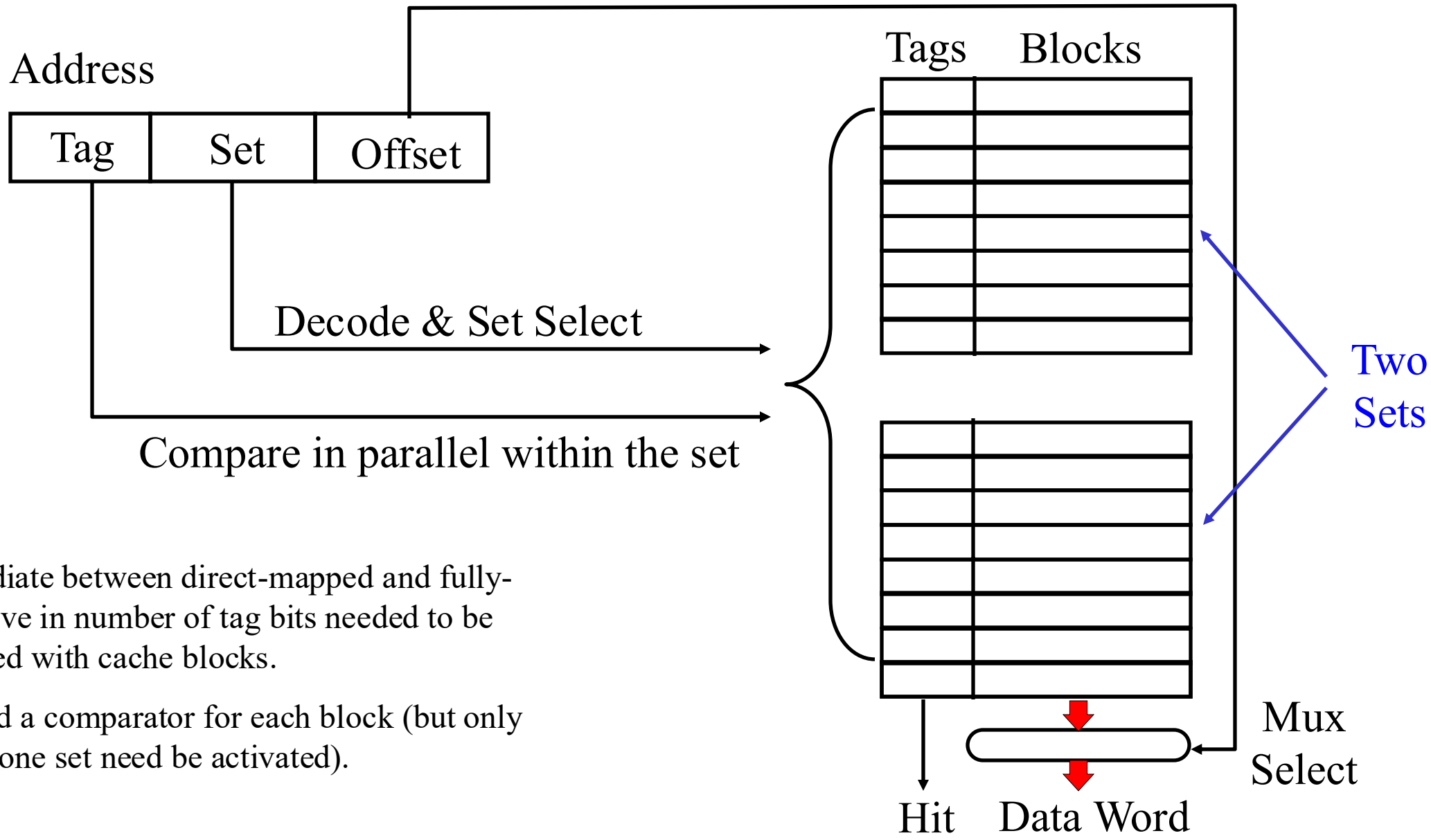
Compared to Direct Cache:

- More address bits (tag) have to be stored with each block.
- A comparator is needed for *each* block, to do the parallel associative lookup.

Set-Associative Placement

- The block address determines not a single frame, but a *frame set* (several frames, grouped together).
  - Frame set # = Block address **mod** # of frame sets
- The block can be placed associatively anywhere within that frame set.
- If there are *n* frames in each frame set, the scheme is called "*n*-way set-associative".
- Direct mapped = 1-way set-associative.
- Fully associative: There is only 1 frame set.

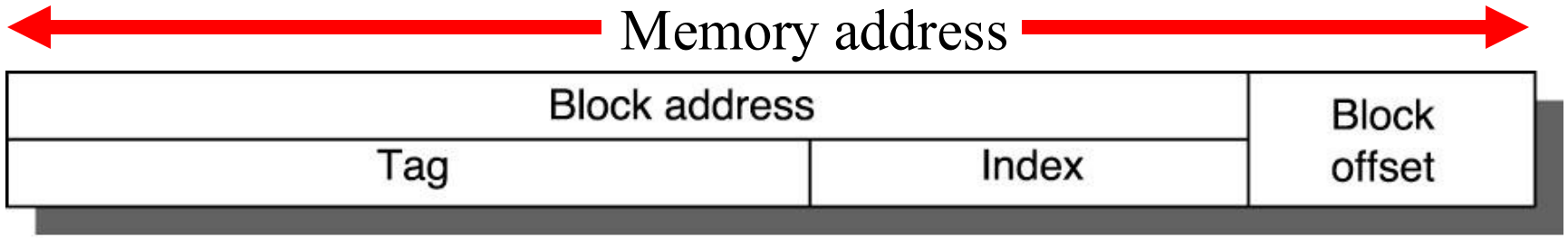
Set-Associative Identification



- Intermediate between direct-mapped and fully-associative in number of tag bits needed to be associated with cache blocks.
- Still need a comparator for each block (but only those in one set need be activated).

Cache Size Equation

- Simple equation for the size of a cache:
  - (Cache size) = (Block size) × (Number of sets) × (Set Associativity)
- Can relate to the size of various address fields:
  - (Block size) = $2^{(\#\text{ of offset bits})}$
  - (Number of sets) = $2^{(\#\text{ of index bits})}$
  - (# of tag bits) = (# of memory address bits) – (# of index bits) – (# of offset bits)



Replacement Strategies

- Which block do we replace when a new block comes in (on cache miss)?
- Direct-mapped: There's only one choice!
- Associative (fully- or set-):
  - If any frame in the set is empty, pick one of those.
  - Otherwise, there are many possible strategies:
    - Random: Simple, fast, and fairly effective
    - Least-Recently Used (LRU)
      - Require bits to record replacement info., e.g. 4-way requires 4! = 24 permutations, need 5 bits to define the MRU to LRU positions
    - FIFO: Replace the oldest block.

Comparison of Replacement Policies

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Data cache misses per 1000 instructions comparing least recently used (LRU), random, and first-in-first-out (FIFO) replacement for several sizes and associativities. There is minor difference between LRU and random for the largest size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using SPECint2000 (gap, gcc, gzip, mcf, and perl) as well as SPECfp2000 (applu, art, equake, lucas, and swim) benchmarks.

Instruction vs. Data Caches

- Instructions and data have different patterns of temporal and spatial locality
  - Also instructions are generally read-only
- Can have *separate* instruction & data caches
  - Advantages
    - Doubles bandwidth between CPU & memory hierarchy
    - Each cache can be optimized for its pattern of locality
  - Disadvantages
    - Slightly more complex design
    - Cannot dynamically adjust cache space taken up by instructions vs. data

Instruction/Data Split and Unified Caches

Size	I-Cache	D-Cache	Unified Cache
8KB	8.16	44.0	63.0
16KB	3.82	40.9	51.0
32KB	1.36	38.4	43.3
64KB	0.61	36.9	39.4
128KB	0.30	35.3	36.2
256KB	0.02	32.6	32.9

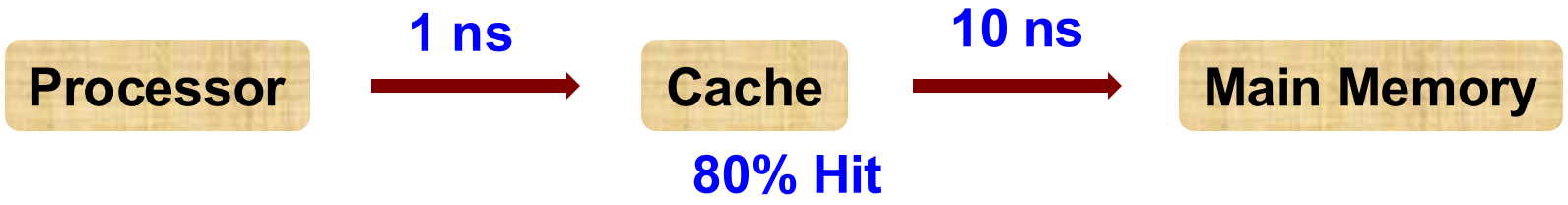
- Miss per 1000 accesses
- Much lower instruction miss rate than data miss rate
- Separate cache presents a trade-off
  - Removes conflict misses between instruction and data blocks
  - Fixes the cache boundary (no sharing)

Review of Memory Hierarchy

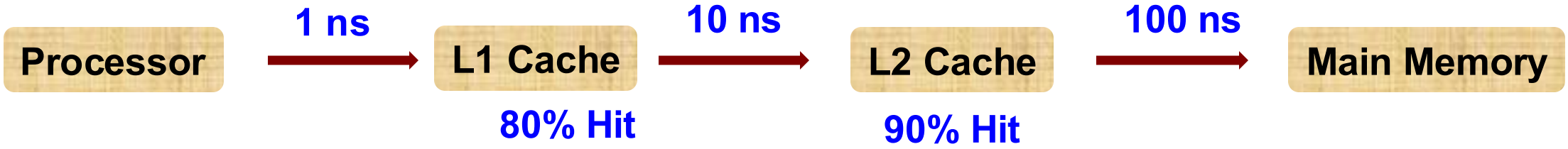
- Introduction
- Cache Basics
- Cache Performance
- Six Basic Cache Optimizations
- Virtual Memory
- Conclusion

Cache Performance

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



Average memory access time = 1+ 0.2 × 10 = 3.0 ns



Average memory access time = 1+ 0.2 × (10 + 0.1 × 100)

= 1+ 0.2 × 20 = 5.0 ns

Cache Performance Equations

- Memory stalls per program (blocking cache):

$$MemoryStallCycle = IC \times (\frac{MemoryAccesses}{Instruction}) \times MissRate \times MissPenalty$$

$$MemoryStallCycle = IC \times (\frac{Misses}{Instruction}) \times MissPenalty$$

- CPU time formula:

$$CPU\ Time = IC \times (CPI_{Execu} + \frac{Memory\ Stall\ Cycle}{Instruction}) \times Cycle\ Time$$

- More cache performance will be given later!

Cache Performance Example

- Ideal CPI=2.0, memory references / inst=1.5, cache size=64KB, miss penalty=75ns, hit time=1 clock cycle
- Compare performance of two caches:
  - Direct-mapped (1-way): cycle time=1ns, miss rate=1.4%
  - 2-way: cycle time=1.25ns, miss rate=1.0%

$$Miss\ Penalty_{1-way} = \left\lceil \frac{75ns}{1ns} \right\rceil = 75\ Cycles$$

$$Miss\ Penalty_{2-way} = \left\lceil \frac{75ns}{1.25ns} \right\rceil = 60\ Cycles$$

$$CPU\ Time_{1-way} = IC \times (2.0 + 1.4\% \times 1.5 \times 75) \times 1ns = 3.575 \times IC$$

$$CPU\ Time_{2-way} = IC \times (2.0 + 1\% \times 1.5 \times 60) \times 1.25ns = 3.625 \times IC$$

Out-Of-Order Processor

- Define new "miss penalty" considering overlap

$$\frac{Mem.Stall\ Cycles}{Instruction} = \frac{Misses}{Instruction} \times (Total\ miss\ latency - Overlap\ miss\ latency)$$

  - Compute *memory latency* and *overlapped latency*

- Example (from previous slide)
  - Assume 30% of 75ns penalty can be overlapped, but with longer (1.25ns) cycle on 1-way design due to Out-of-Order

$$Miss\ Penalty_{1-way} = \left\lceil \frac{75ns \times 70\%}{1.25ns} \right\rceil = 42\ Cycles$$

$$CPU\ Time_{1-way,OOO} = IC \times (2.0 + 1.4\% \times 1.5 \times 42) \times 1.25ns = 3.60 \times IC$$

Cache Performance Improvement

$$\text{Average memory access time} = (\text{Hit time}) + \underbrace{(\text{Miss rate}) \times (\text{Miss penalty})}_{\text{“Amortized miss penalty”}}$$

- Reduce miss penalty:
  - Multilevel cache; Critical word first and early restart; priority to read miss; Merging write buffer; Victim cache
- Reduce miss rate:
  - Larger block size; Increase cache size; Higher associativity; Way prediction and Pseudo-associative caches; Compiler optimizations
- Reduce miss penalty/rate via parallelism:
  - Non-blocking cache; Hardware and Compiler-controlled prefetching
- Reduce hit time:
  - Small simple cache; Avoid address translation in indexing cache; Pipelined cache access; Trace caches

Review of Memory Hierarchy

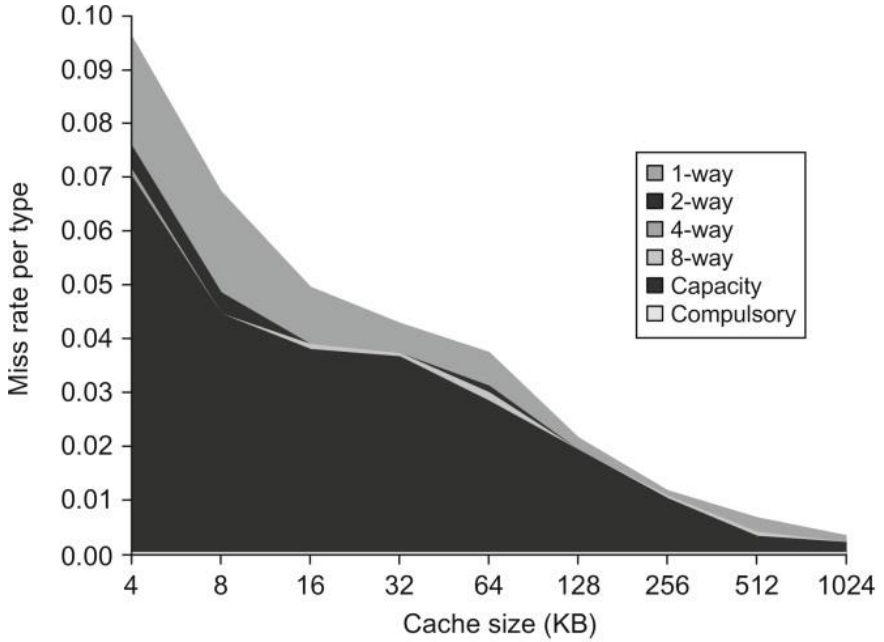
- Introduction
- Cache Basics
- Cache Performance
- Six Basic Cache Optimizations
- Virtual Memory
- Conclusion

Three Types of Misses

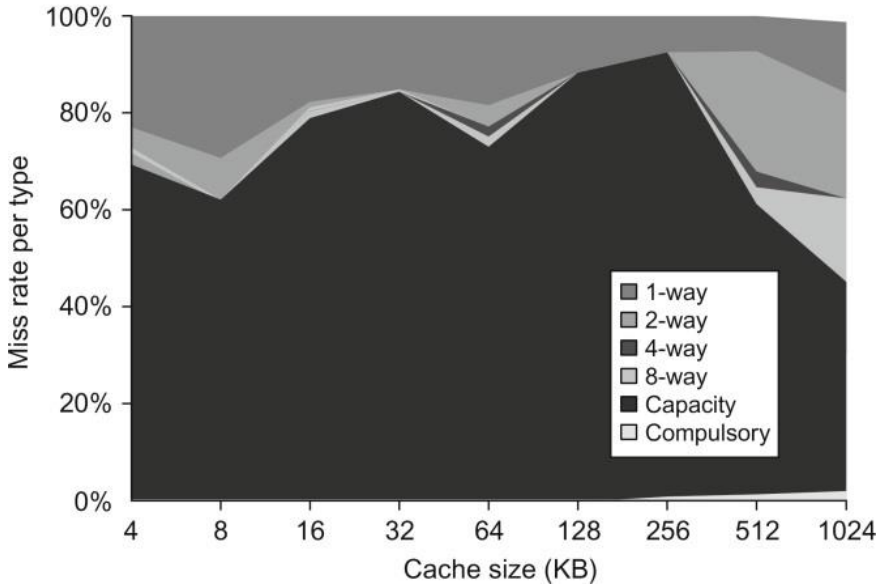
- Compulsory
  - During a program, the very first access to a block will not be in the cache (unless pre-fetched)
- Capacity
  - The *working set* of blocks accessed by the program is too large to fit in the cache
- Conflict
  - Unless cache is fully associative, sometimes blocks may be evicted too early because too many frequently-accessed blocks map to the same limited set of frames/sets.

Miss Rate Analysis

Total Miss Rate



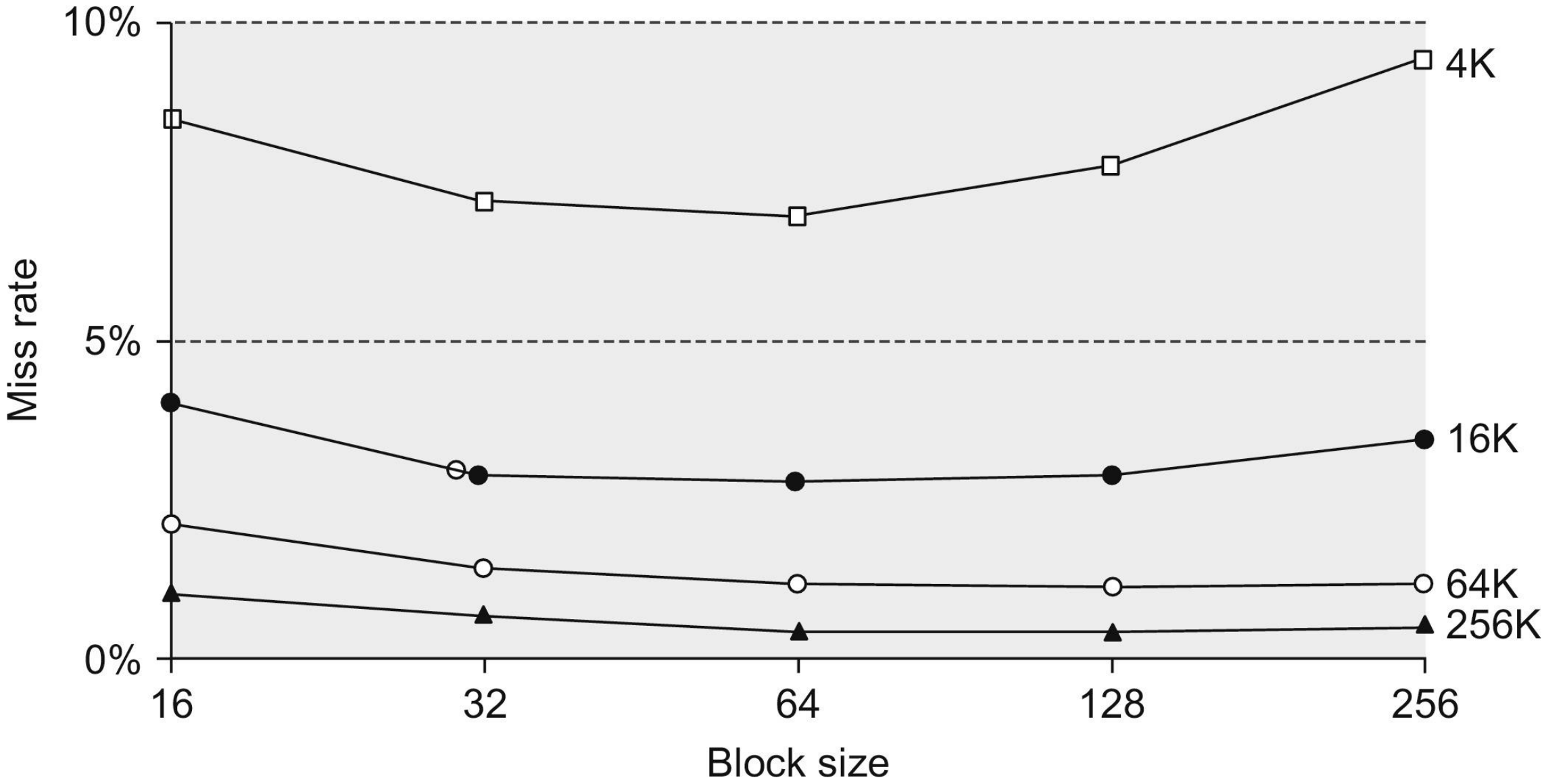
Miss Rate Distribution



First Optimization: Larger Block Size

- Keep cache size & associativity constant
- Reduces compulsory misses
  - Due to spatial locality
  - More accesses are to a pre-fetched block
- Increases capacity misses
  - More unused locations pulled into cache
- May increase conflict misses (slightly)
  - Fewer sets may mean more blocks utilized per set
  - Depends on pattern of addresses accessed
- Increases miss penalty - longer block transfers

Block Size Effect



Miss rate goes up if the block is too large relative to the cache size

Block Size vs. Miss Rate

	Cache Size			
Block Size	**4K**	**16K**	**64K**	**256K**
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	**9.51%**	3.29%	1.15%	**0.49%**

- In 4K cache, 256 bytes block has highest miss rate
- 256 bytes block becomes profitable for 256K cache

Second Optimization: Larger Caches

- Keep block size, set size, *etc.* constant
- No effect on compulsory misses.
  - Block still won't be there on its 1st access!
- Reduces capacity misses
  - More capacity!
- Reduces conflict misses (in general)
  - Working blocks spread out over more frame sets
  - Fewer blocks map to a set on average
  - Less chance that the number of active blocks that map to a given set exceeds the set size.
- But, increases hit time! (and cost.)

Block Size vs. Avg. Access Time

		Cache Size			
Block Size	**Miss Penalty**	**4K**	**16K**	**64K**	**256K**
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

- Block sizes 32 and 64 perform better

Third Optimization: Higher Associativity

- Keep cache size & block size constant
  - Decreasing the number of sets
- No effect on compulsory misses
- No effect on capacity misses
  - By definition, these are misses that would happen anyway in fully-associative
- Decreases conflict misses
  - Blocks in active set may not be evicted early
- Can increase hit time (slightly)
  - Direct-mapped is fastest
  - *n*-way associative lookup a bit slower for larger *n*

Performance Comparison

- Assume:

$$Hit\ Time_{1-way} = Cycle\ Time_{1-way} \qquad Miss\ Penalty = 25 \times Cycle\ Time_{1-way}$$
$$Hit\ Time_{2-way} = 1.36 \times Cycle\ Time_{1-way}$$
$$Hit\ Time_{4-way} = 1.44 \times Cycle\ Time_{1-way}$$
$$Hit\ Time_{8-way} = 1.52 \times Cycle\ Time_{1-way}$$

- 4KB, 1-way miss-rate=9.8%; 4-way miss-rate=7.1%

$$Average\ Mem\ Access\ Time_{1-way} = 1.00 + Miss\ Rate \times 25$$
$$= 1.00 + 0.098 \times 25 = 3.45$$

$$Average\ Mem\ Access\ Time_{4-way} = 1.44 + Miss\ Rate \times 25$$
$$= 1.44 + 0.071 \times 25 = 3.215$$

Higher Set-Associativity

Cache Size	1-way	2-way	4-way	8-way
4KB	3.44	3.25	3.22	3.28
8KB	2.69	2.58	2.55	2.62
16KB	2.23	2.40	2.46	2.53
32KB	2.06	2.30	2.37	2.45
64KB	1.92	2.14	2.18	2.25
128KB	1.52	1.84	1.92	2.00
256KB	1.32	1.66	1.74	1.82
512KB	1.20	1.55	1.59	1.66

- Higher associativity increases the cycle time
- The table shows the average memory access time
  - 1-way is better in most of the cases

Fourth Optimization: Multi-Level Caches

- What is important?
  - faster caches?
  - or larger caches?
- Average memory access time
  - Hit time (L1) + Miss rate (L1) x Miss Penalty (L1)
- Miss penalty (L1)
  - Hit time (L2) + Miss rate (L2) x Miss Penalty (L2)
- Can plug 2nd equation into the first:
  - Average memory access time
    - Hit time(L1) + Miss rate(L1) x
      (Hit time(L2) + Miss rate(L2)x Miss penalty(L2))

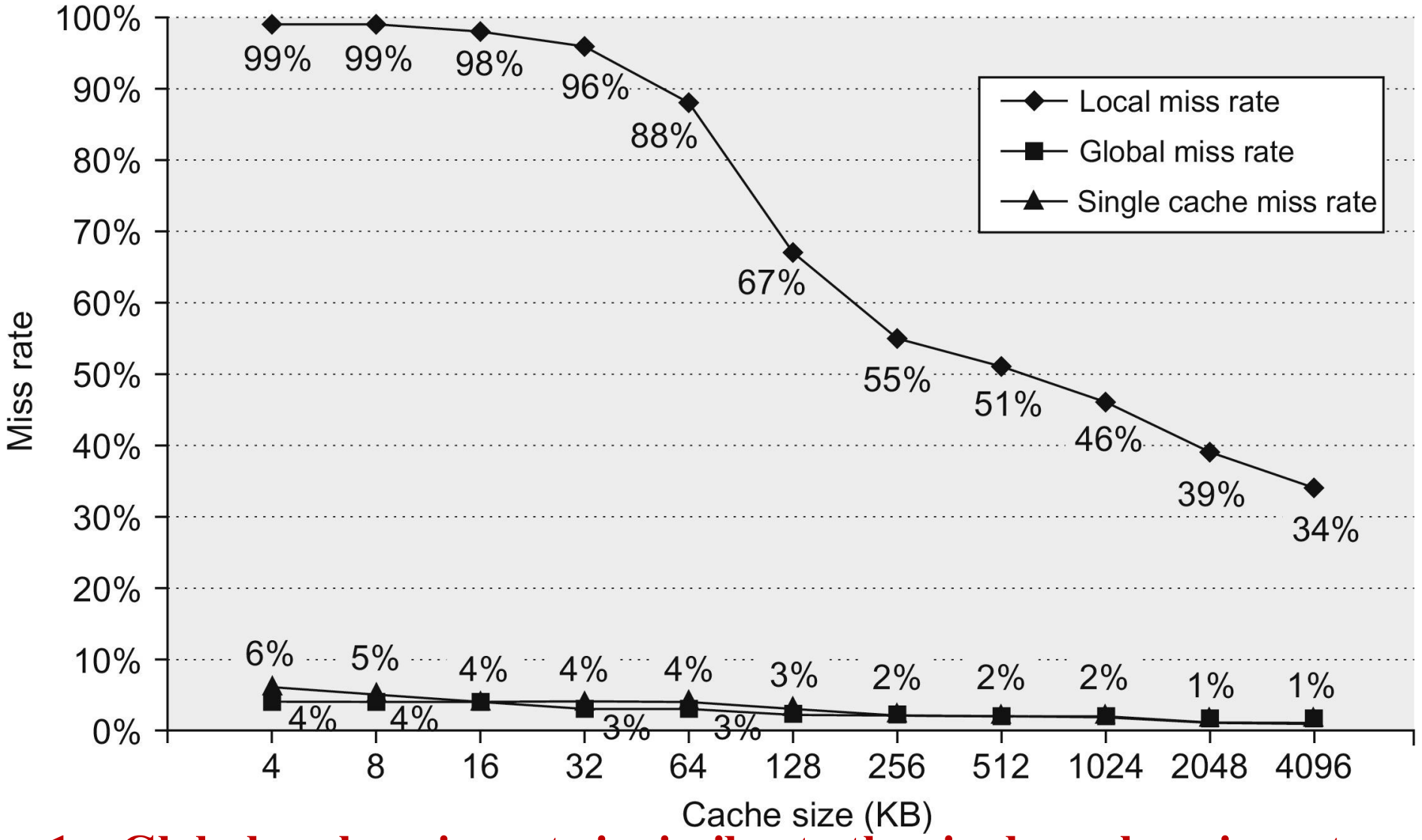
Multi-level Cache Terminology

- "Local miss rate"
  - The miss rate of one hierarchy level by itself.
  - # of misses at that level / # accesses to that level
  - e.g. Miss rate(L1), Miss rate(L2)
- "Global miss rate"
  - The miss rate of a whole group of hierarchy levels
  - # of accesses coming out of that group (to lower levels) / # of accesses to that group
  - This is the product of the miss rates at each level in the group.
  - Global L2 Miss rate = Miss rate(L1) × Local Miss rate(L2)

Effect of 2-level Caching

- L2 size usually much bigger than L1
  - Provide reasonable hit rate
  - Decreases miss penalty of 1st-level cache
  - May increase L2 miss penalty
- Multiple-level cache inclusion property
  - Inclusive cache: L1 is subset of L2; simplify cache coherence mechanism, effective cache size = L2
  - Exclusive cache: L1, L2 are exclusive; increase effect cache sizes = L1 + L2
  - Enforce inclusion property: Backward invalidation on L2 replacement

L2 Cache Performance



1. **Global cache miss rate is similar to the single cache miss rate**
2. **Local miss rate is not a good measure of secondary caches**

5th Optimization: Read Misses Take Priority

- Processor must wait on a read, not on a write
  - Miss penalty is higher for reads to begin with and more benefit from reducing read miss penalty
- Write buffer can queue values to be written
  - Until memory bus is not busy with reads
  - Careful about the memory consistency issue
- What if we want to read a block in write buffer?
  - Wait for write, then read block from memory
  - Better: Read block out of write buffer.
- Dirty block replacement when reading
  - Write old block, read new block - Delays the read.
  - Old block to buffer, read new, write old. - Better!

6th Optimization: Avoid Address Translation

- In systems with virtual address spaces, virtual address must be mapped to physical addresses.
- If cache blocks are indexed/tagged with physical addresses, we must do this translation before we can do the cache lookup. *Long hit time!*
- Solution: Access cache using the *virtual* address. Call this a "Virtual Cache"
  - Drawback: Cache flush on context switch
    - Can fix by tagging blocks with Process Ids (PIDs)
  - Another problem: "Aliasing", i.e., two virtual addresses mapped to same real address
    - Fix with anti-aliasing or page coloring

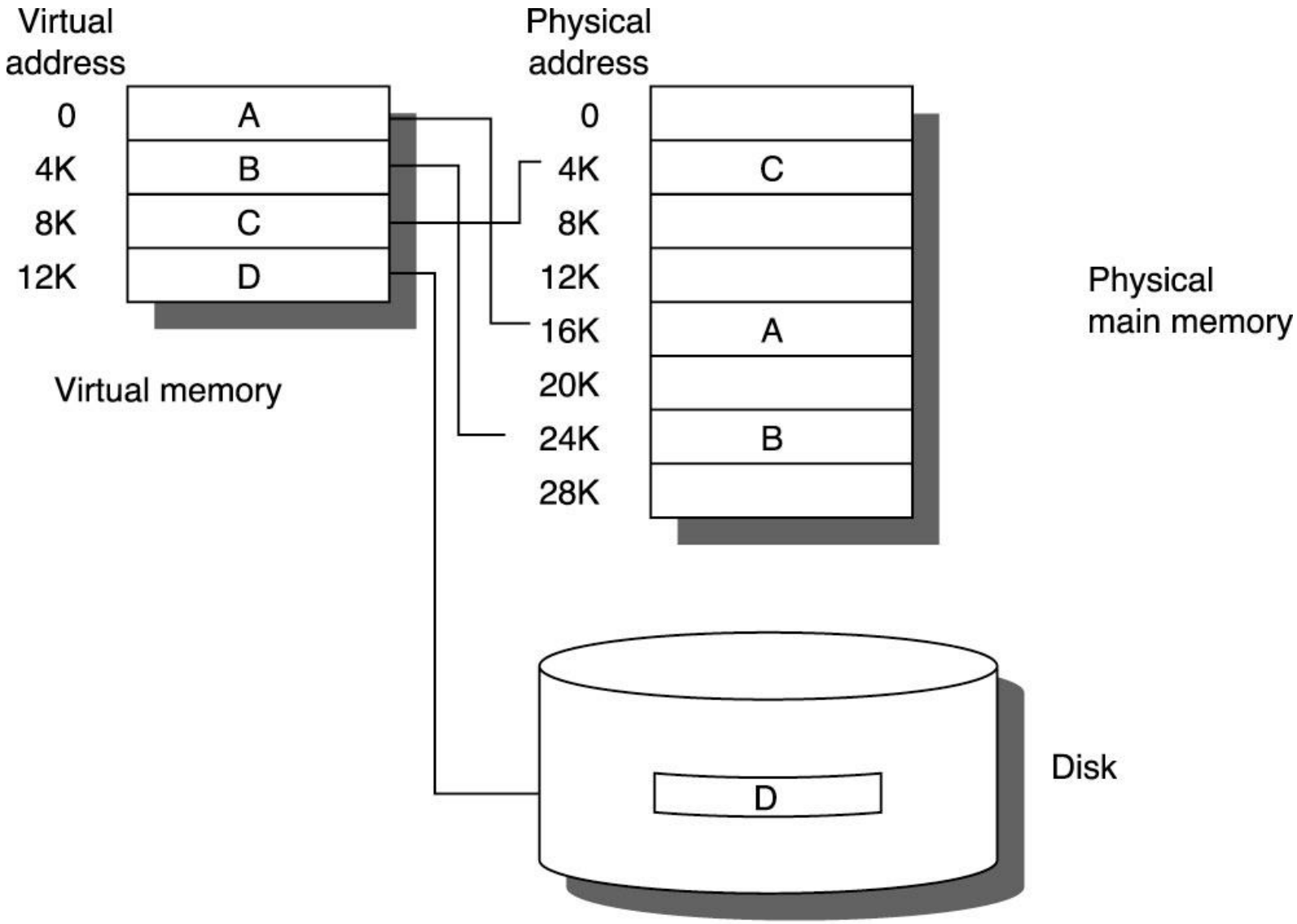
Review of Memory Hierarchy

- Introduction
- Cache Basics
- Cache Performance
- Six Basic Cache Optimizations
- Virtual Memory
- Conclusion

Why Virtual Memory?

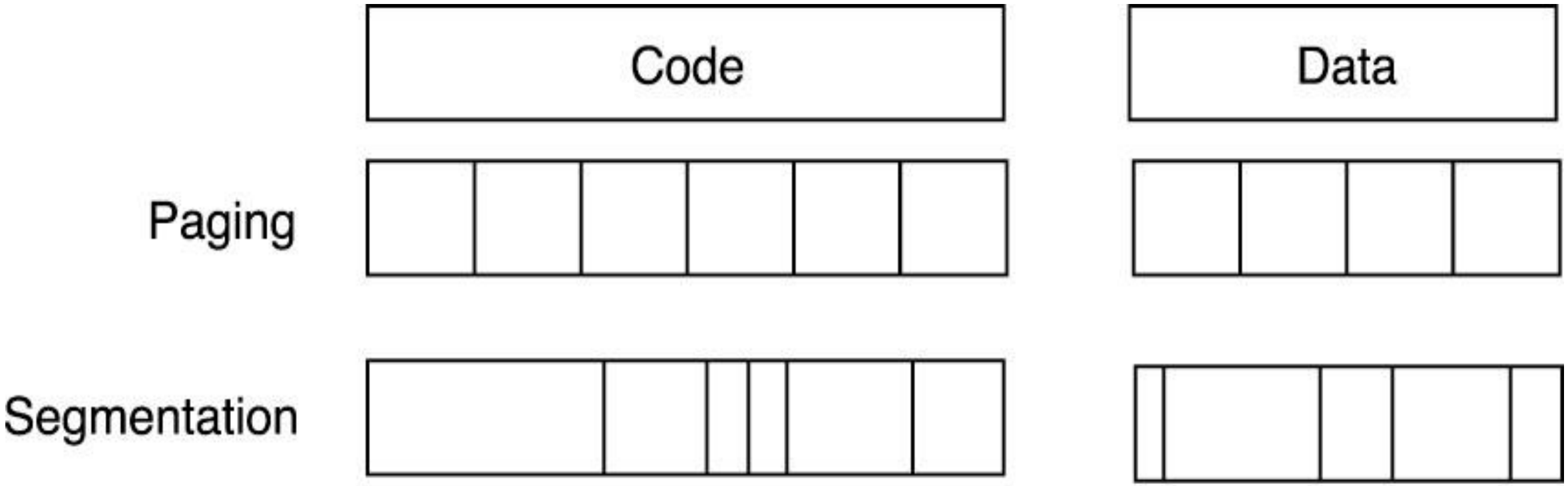
- Sharing a smaller amount of physical memory
  - Each process use a small part of address space.
  - Needs a protection mechanism.
- Virtual memory was invented to automatically manage two levels of memory hierarchy
  - Old days, the main memory used to be small and program used to be big
  - The programmer used to divide the programs into several parts and load into memory so that the part does not access outside physical main memory.
- It also enables relocation.

Virtual Memory



The addition of the virtual memory mechanism complicated the cache access

Paging vs. Segmentation



- *Paged Segment*: Each segment has integral number of pages for easy replacement and can still treat each segmentation as a unit

Paging versus Segmentation

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Invisible to application programmer	May be visible to application programmer
Memory use inefficiently	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main mem.)
Efficient disk traffic	Yes - adjust page size to balance access/transfer time	Not always (small segments may transfer just a few bytes)

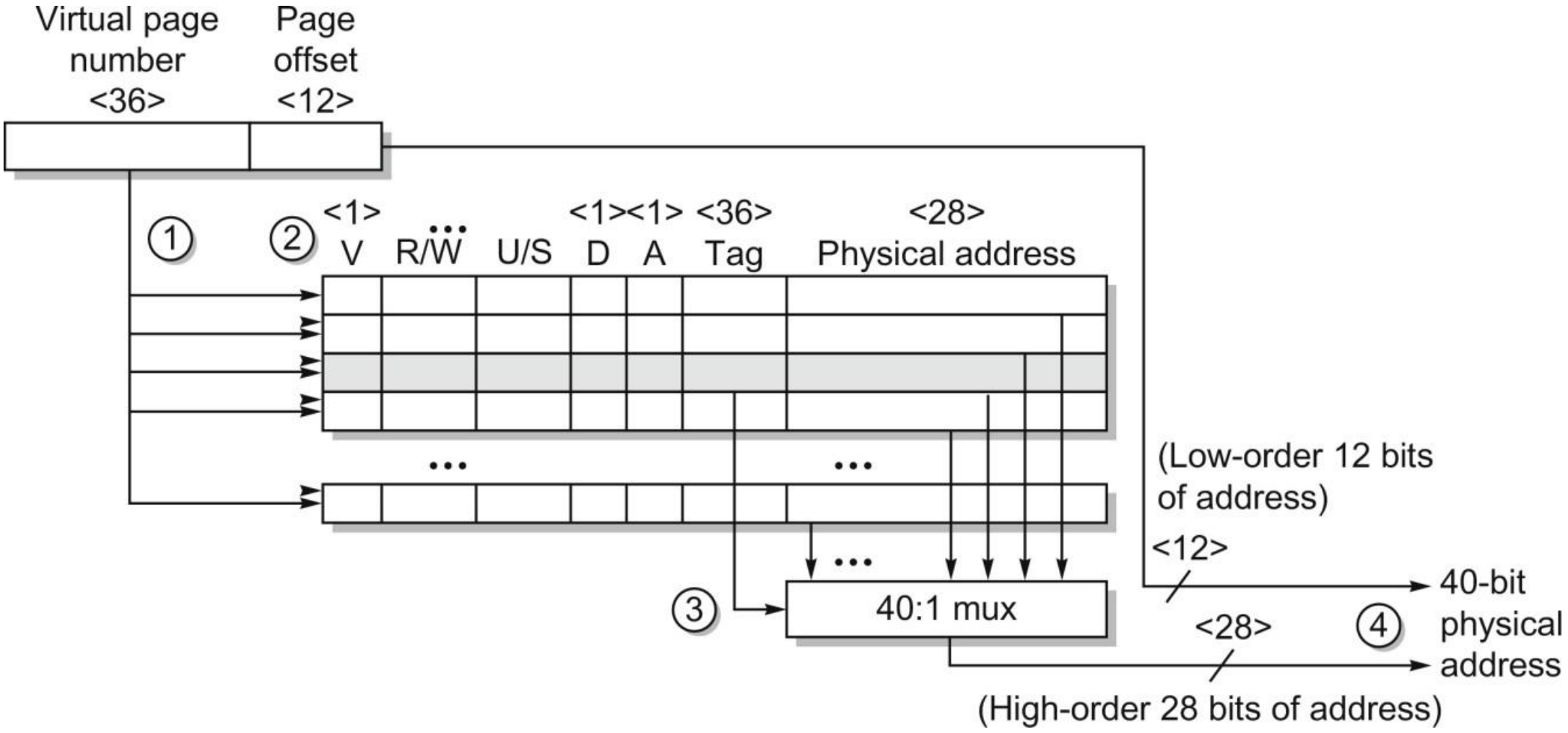
Four Important Questions

- Where to place a block in main memory?
  - Operating systems takes care of it
  - Replacement takes very long – fully associative
- How to find a block in main memory?
  - Page table is used
    - Offset is concatenated when paging is used
    - Offset is added when segmentation is used.
- Which block to replace when needed?
  - Obviously – LRU is used to minimize page faults
- What happens on a write?
  - Magnetic disks takes millions of cycles to access.
  - Always write back (use of dirty bit).

Fast Address Calculation

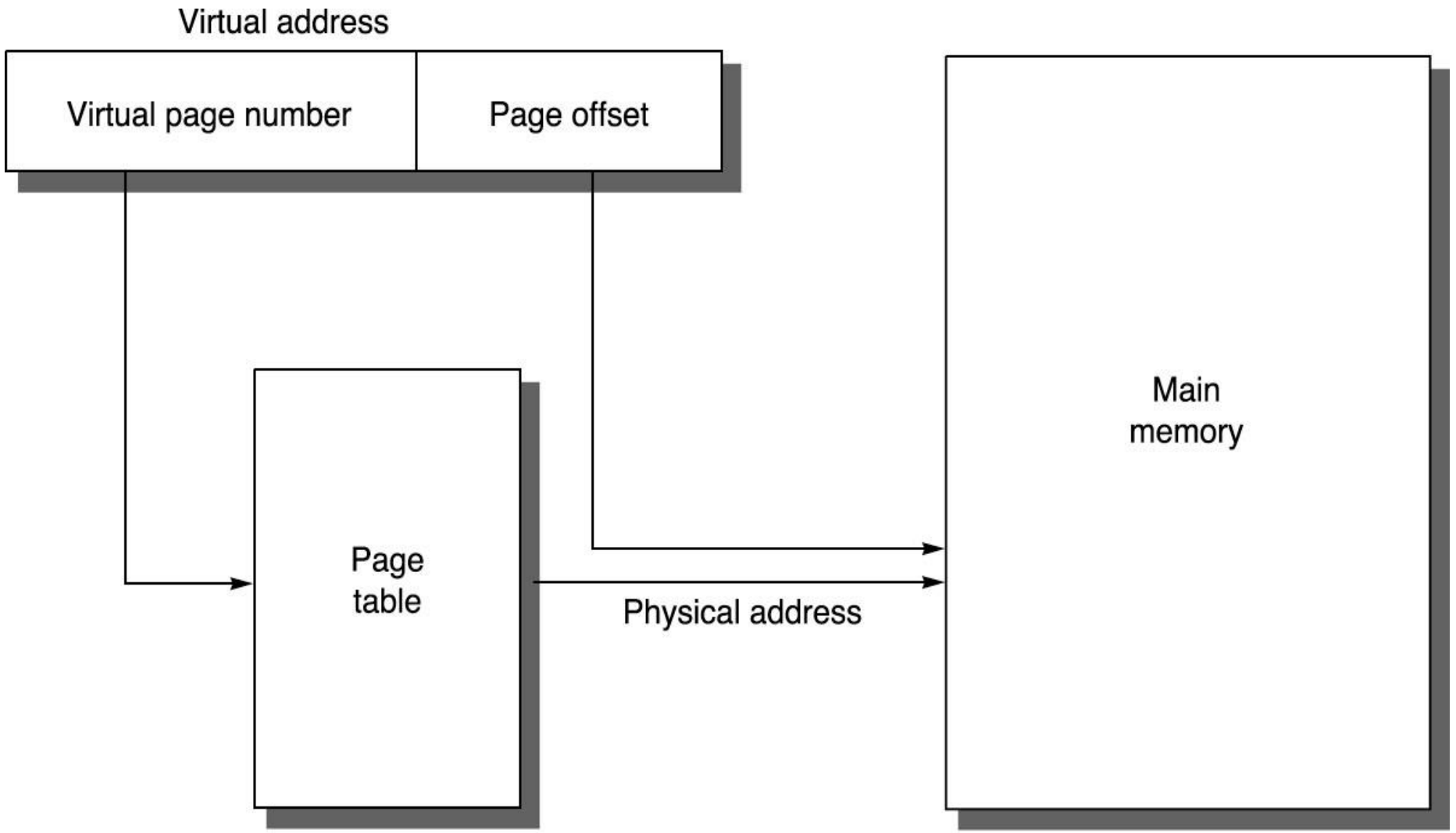
- Page tables are very large
  - Kept in main memory
    - Two memory accesses for one read or write
- Remember the last translation
  - Reuse if the address is on the same page
- Exploit the principle of locality
  - If access have locality, the address translations should also have locality
  - Keep the address translations in a cache
    - Translation lookaside buffer (TLB)
    - The tag part stores the virtual address and the data part stores the page number.

TLB Example: AMD Opteron



Operation of the Opteron data TLB during address translation. The four steps of a TLB hit are shown as *circled numbers*. This TLB has 40 entries. Section B.5 describes the various protection and access fields of an Opteron page table entry.

Addressing Virtual Memories



Virtual to Physical Address Mapping

- Physical address = physical page number + page offset
- Page Table ➔

Index	Value
0	
1	1
2	
3	0

- Page Size
  - 1024 bytes

Virtual Address	Physical Address
123	Miss
1234	1234
2435	Miss
3456	384

Example 1:

1234 / 1024 ➔ quotient **1**, remainder 210

Virtual Page **1** and offset of 210

➔ ➔ ➔

Physical page **1** and offset 210

1 * 1024 + 210 ➔ 1234

Example 2:

3456 / 1024 ➔ quotient **3**, remainder 384

Virtual Page **3** and offset of 384

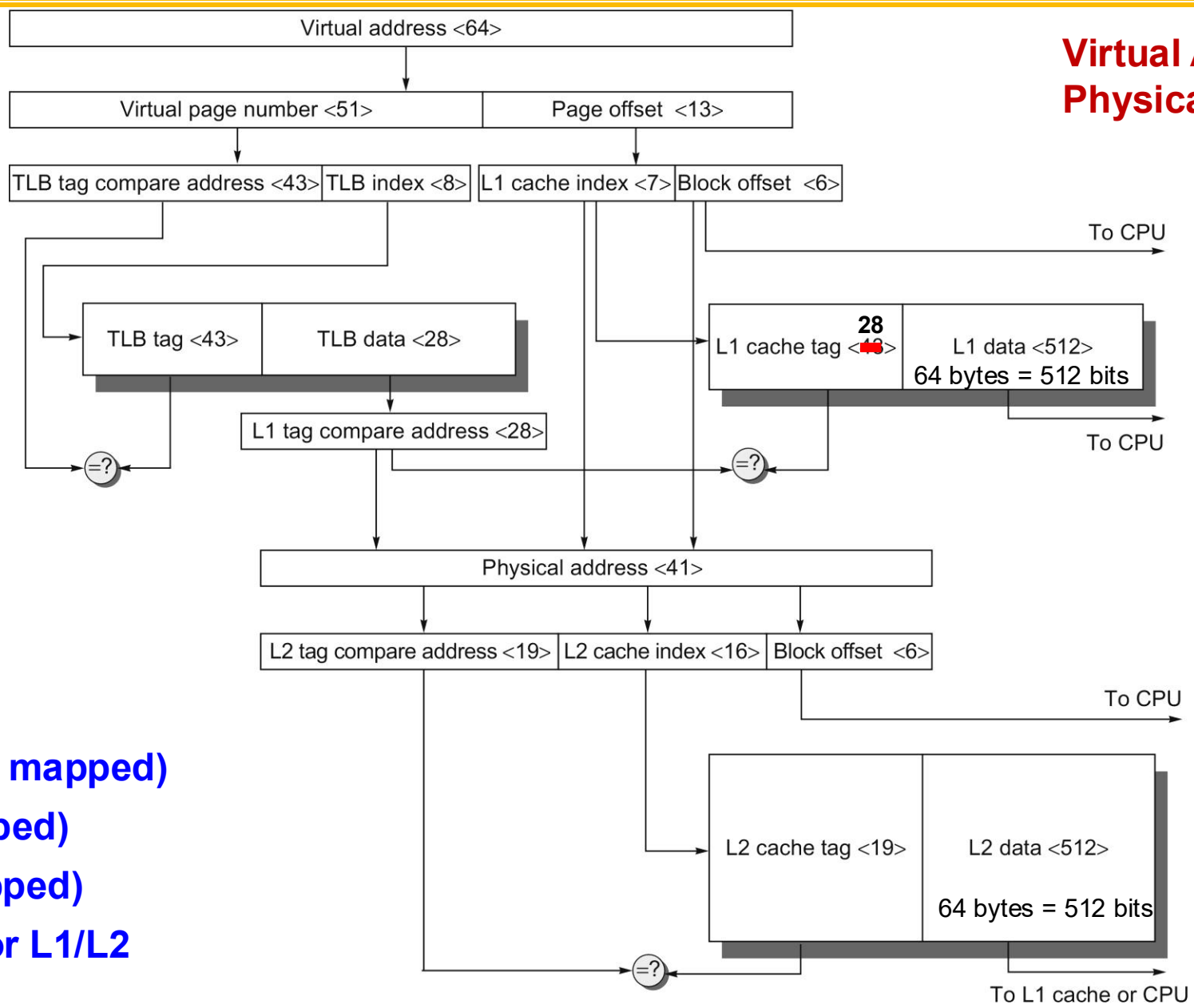
➔ ➔ ➔

Physical page **0** and offset 384

0 * 1024 + 384 ➔ 384

A Memory Hierarchy Example

Virtual Address: 64 bits
Physical Address: 41 bits



8 KB Pages

256 entry TLB (direct mapped)

L1 – 8KB (direct mapped)

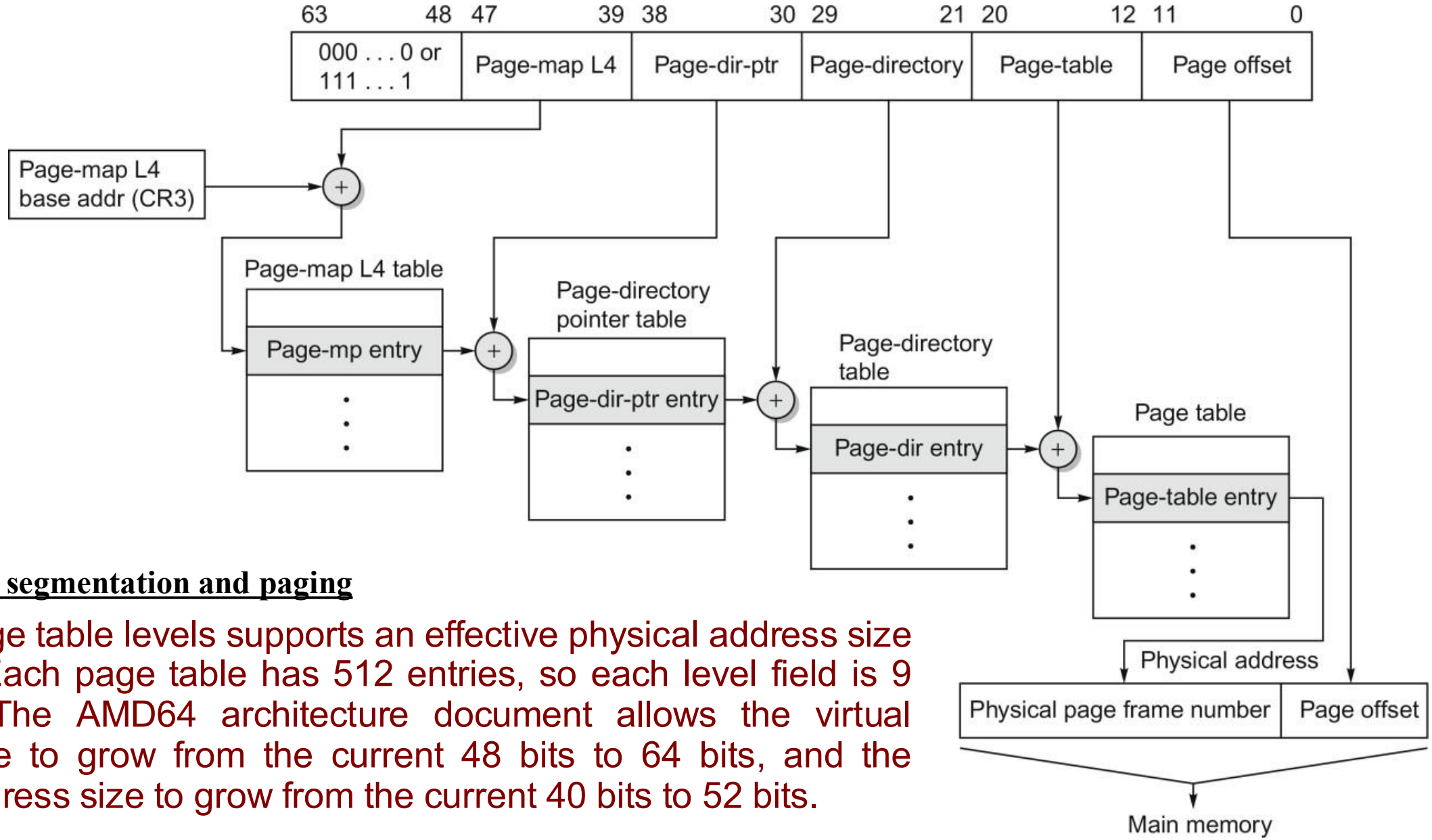
L2 – 4 MB (direct mapped)

Block size 64 bytes for L1/L2

Protection of Virtual Memory

- Maintain two registers
  - Base
  - Bound
- For each address check
  - base <= address <= bound
- Provide two modes
  - User
  - OS (kernel, supervisor, executive)

Opteron Virtual Address Mapping



Supports both segmentation and paging

The four page table levels supports an effective physical address size of 40 bits. Each page table has 512 entries, so each level field is 9 bits wide. The AMD64 architecture document allows the virtual address size to grow from the current 48 bits to 64 bits, and the physical address size to grow from the current 40 bits to 52 bits.

Conclusion

- Basics of three cache designs
  - Direct Cache
  - Set-Associative Cache
  - Fully Associative Cache
- Computation of average memory access time
- Optimizations to improve cache performance
- Virtual memory and address translation
- Please read Section B.1 – B.4
- We review design of memory hierarchy next.

Pop Quiz

- Page Size = 1024 bytes = 1 KB
- Page Table
  - **Index** is the virtual page number
  - **Value** is the physical page number

Index	Value
0	3
1	2
2	1
3	0