

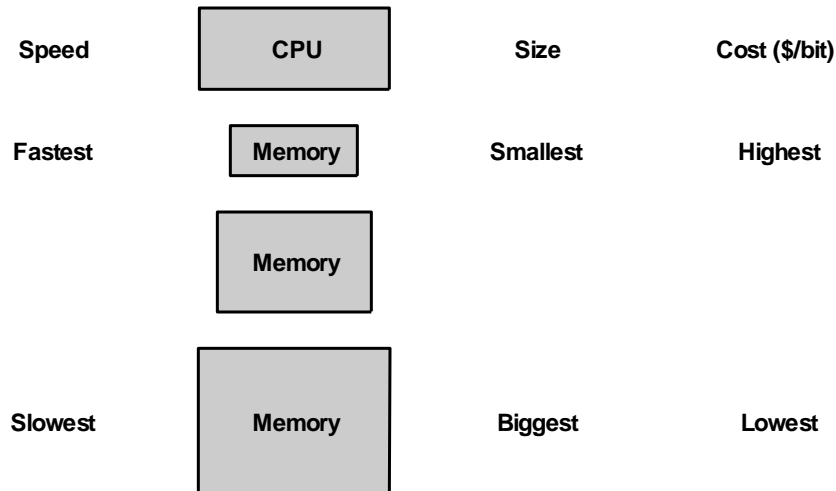
# Exploiting Memory Hierarchy

# Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, \$2000 – \$5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, \$20 – \$75 per GB
- Magnetic disk
  - 5ms – 20ms, \$0.20 – \$2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

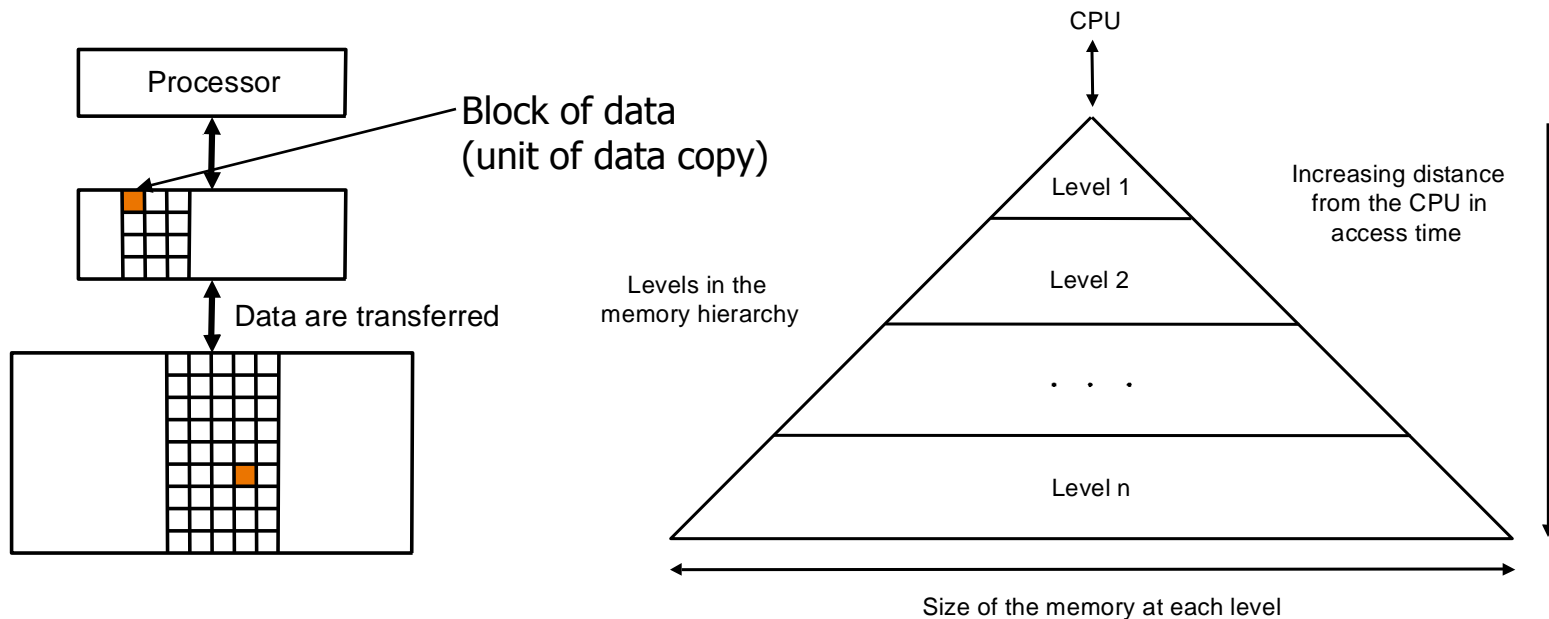
# Memories: Review

- *DRAM* (Dynamic Random Access Memory):
  - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
  - very small – 1 transistor per bit – but factor of **5 to 10 slower** than SRAM
  - used for *main memory*
- *SRAM* (Static Random Access Memory):
  - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
  - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
  - used for *cache*



# Memory Hierarchy

- Users want large and fast memories...
  - expensive and they don't like to pay...
- Make it seem like they have what they want...
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*



# Locality

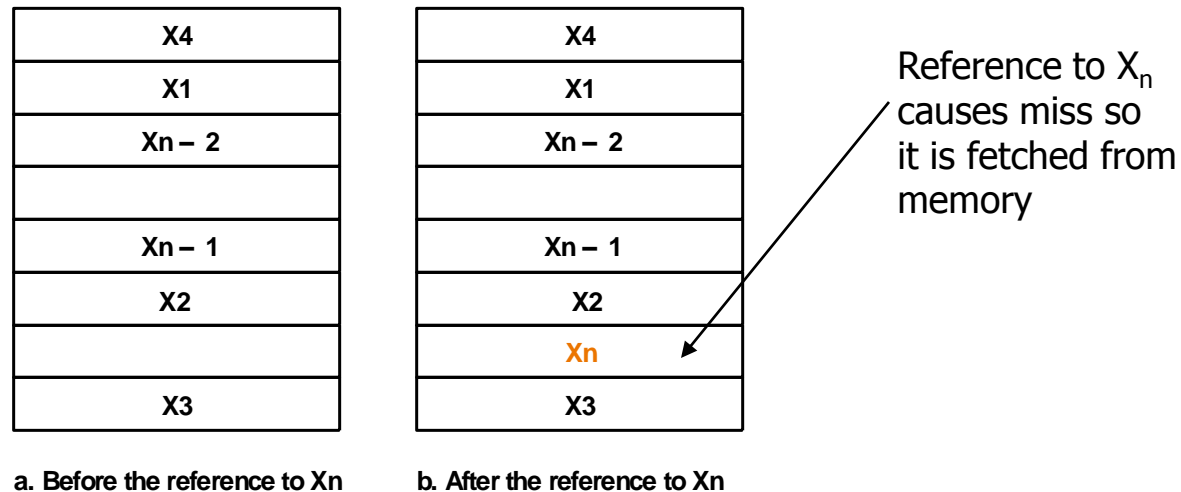
- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
  - *spatial locality*: *nearby items* will tend to be referenced soon

# Hit and Miss

- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is in upper level
  - *miss*: data requested is not in upper level
  - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
  - *miss rate*: fraction of memory accesses that are not hits
    - $\text{miss rate} = 1 - \text{hit rate}$
  - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

# Caches

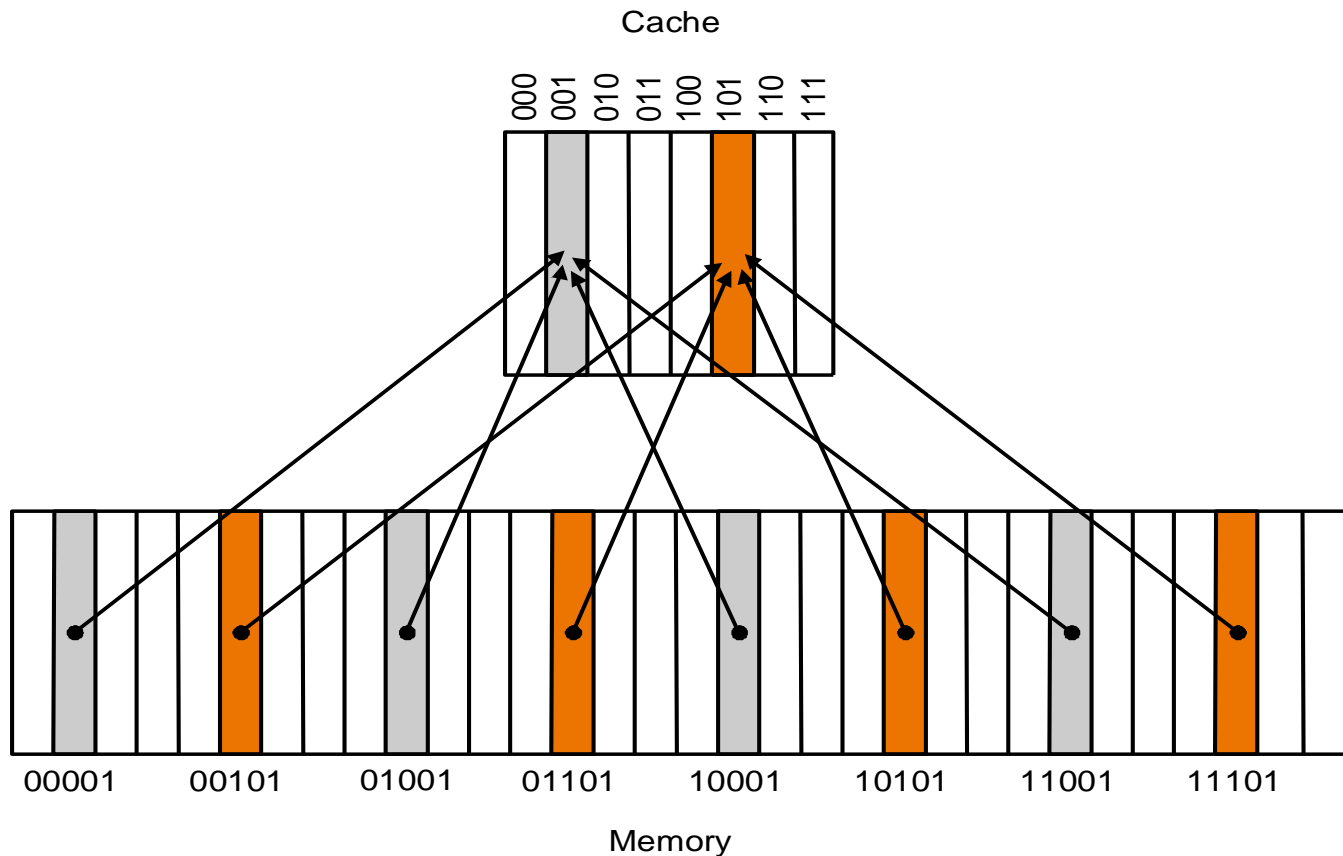
- By simple example
  - assume block size = one word of data



- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme...*

# Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address *mod* cache size (*unique*)
  - if cache size =  $2^m$ , cache address = lower  $m$  bits of  $n$ -bit memory address
  - remaining upper  $n-m$  bits kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry





# Accessing Cache

- Example:

(0) Initial state:

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

(1) Address referred 10110 (*miss*):

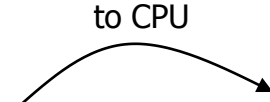
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(2) Address referred 11010 (*miss*):

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		

(3) Address referred 10110 (*hit*):

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		

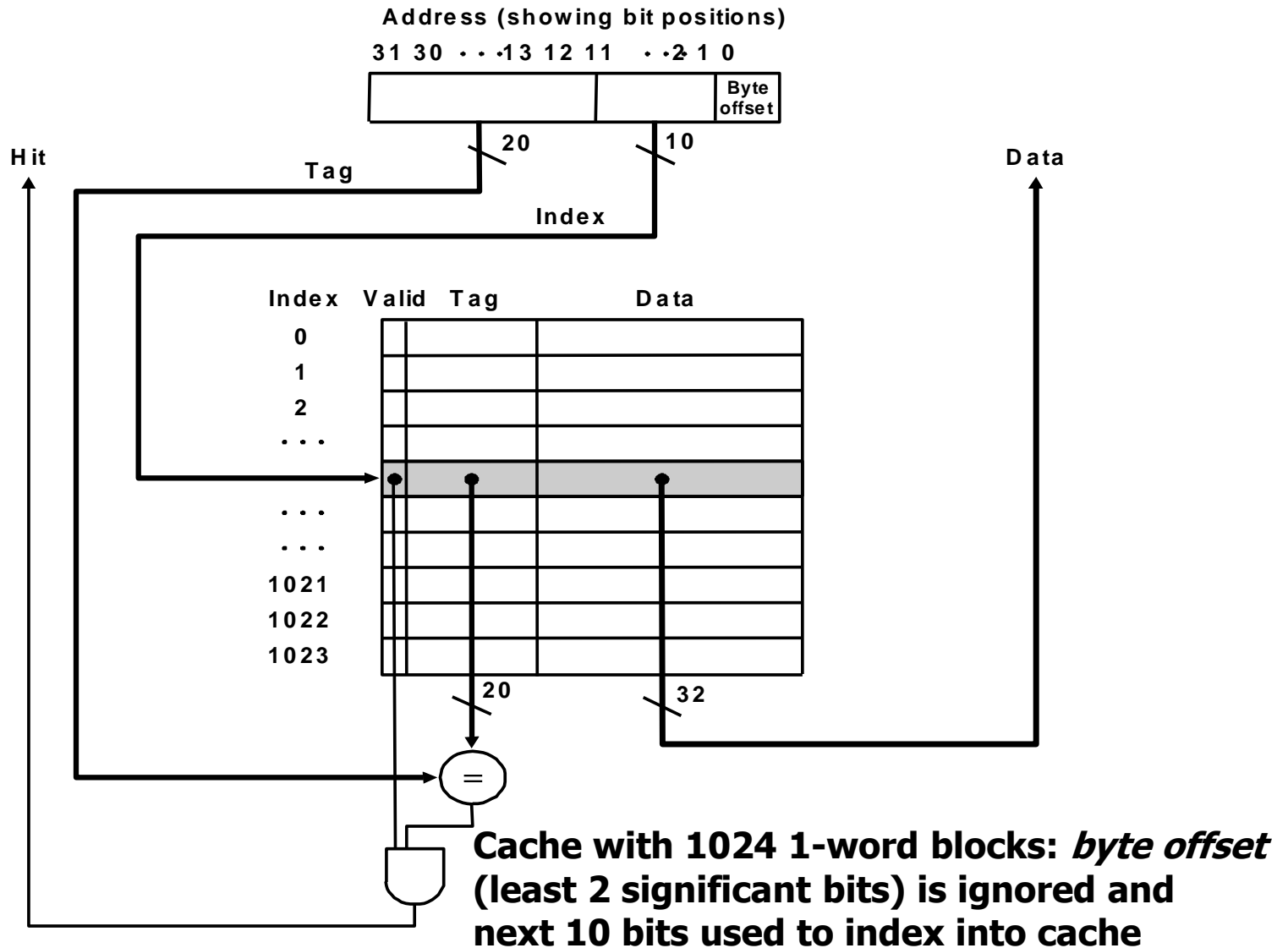


(4) Address referred 10010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

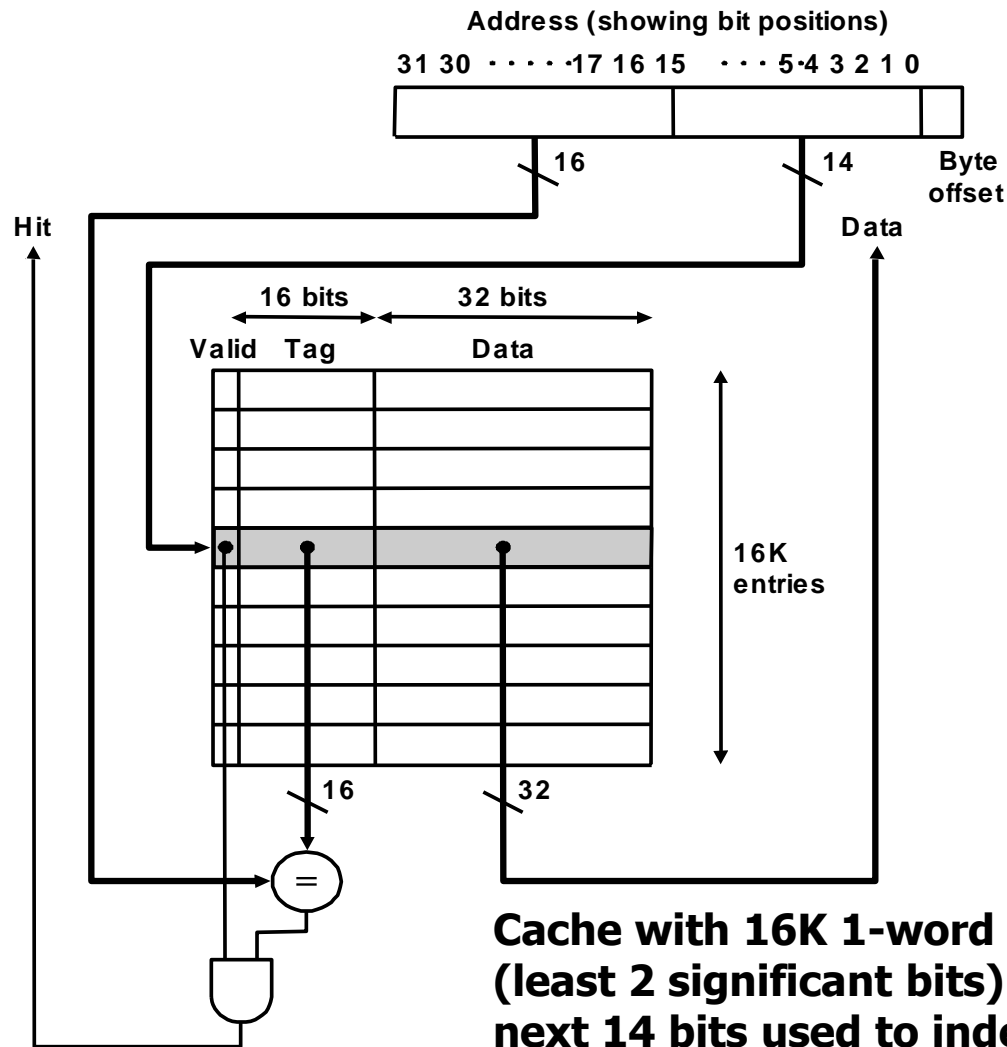
# Direct Mapped Cache

- MIPS style:



*What kind of locality are we taking advantage of?*

# DECStation 3100 Cache(MIPS R2000 processor)



# Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. Send original PC value (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use*

# Cache Write Hit/Miss

## *Write-through scheme*

- on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
- on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
- Write-through is slow because of always required memory write
  - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
  - when a word in the write buffer completes writing into main memory that buffer slot is freed and becomes available for future writes
  - DEC 3100 write buffer has 4 words

## ■ *Write-back scheme*

- write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
- more efficient than write-through, more complex to implement

- Taking advantage of spatial locality with *larger* blocks:



# Direct Mapped Cache: Taking Advantage of Spatial Locality

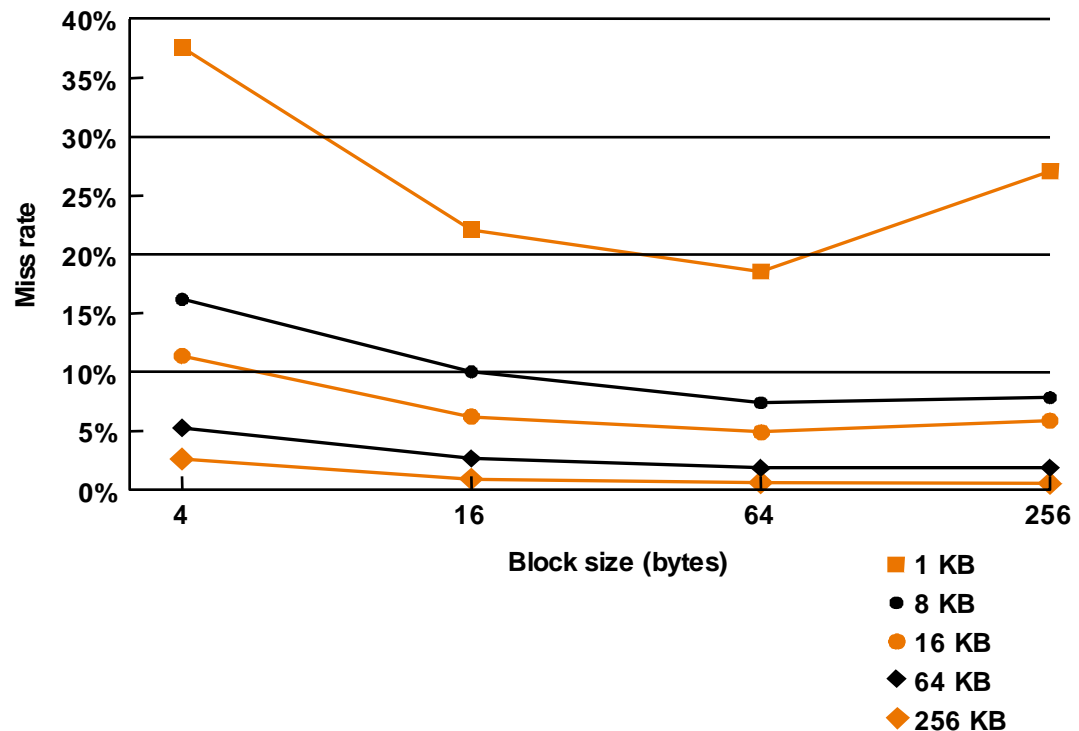
- *Cache replacement* in large (multiword) blocks:
  - word *read miss*: read entire block from main memory
  - word *write miss*: *cannot* simply write word and tag!
  - writing in a *write-through* cache:
    - if *write hit*, i.e., tag of requested address and cache entry are equal, continue as for 1-word blocks by replacing word and writing block to both cache and memory
    - if *write miss*, i.e., tags are unequal, fetch block from memory, replace word that caused miss, and write block to both cache and memory
    - therefore, unlike case of 1-word blocks, a write miss with a multiword block causes a memory read



# Direct Mapped Cache: Taking Advantage of Spatial Locality

- Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because
  - there are few blocks
  - competition for blocks increases (cache **pollution**)
  - blocks get ejected before most of their words are accessed (**thrashing** in cache)

**Miss rate vs. block size for various cache sizes**



# Example

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*
- Cache data = 128 KB =  $2^{17}$  bytes =  $2^{15}$  words =  $2^{15}$  blocks
- Cache entry size = block data bits + tag bits + valid bit  
 $= 32 + (32 - 15 - 2) + 1 = 48$  bits
- Therefore, cache size =  $2^{15} \times 48$  bits =  
 $2^{15} \times (1.5 \times 32)$  bits =  $1.5 \times 2^{20}$  bits = 1.5 Mbits
  - data bits in cache = 128 KB  $\times$  8 = 1 Mbits
  - total cache size/actual cache data = 1.5

# Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*
- Cache size = 128 KB =  $2^{17}$  bytes =  $2^{15}$  words =  $2^{13}$  blocks
- Cache entry size = block data bits + tag bits + valid bit  
=  $128 + (32 - 13 - 2 - 2) + 1 = 144$  bits
- Therefore, cache size =  $2^{13} \times 144$  bits =  
 $2^{13} \times (1.25 \times 128)$  bits =  $1.25 \times 2^{20}$  bits = 1.25 Mbits
  - data bits in cache = 128 KB  $\times$  8 = 1 Mbits
  - total cache size/actual cache data = 1.25

# Example Problem

- *Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?*
- As block size = 16 bytes:  
byte address 1200  $\Rightarrow$  block address  $\lfloor 1200/16 \rfloor = 75$
- As cache size = 64 blocks:  
block address 75  $\Rightarrow$  cache block  $(75 \bmod 64) = 11$



# Block Size Considerations

---

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks  $\Rightarrow$  fewer of them
    - More competition  $\Rightarrow$  increased miss rate
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

# Performance

- Simplified model assuming equal read and write miss penalties:
  - $\text{CPU time} = (\text{execution cycles} + \text{memory stall cycles}) \times \text{cycle time}$
  - $\text{memory stall cycles} = \text{number of memory accesses} \times \text{miss rate} \times \text{miss penalty}$
- Therefore, two ways to improve performance in cache:
  - decrease miss rate
  - decrease miss penalty
  - *what happens if we increase block size?*

# Example

- Assume for a given machine and program:
    - instruction cache miss rate 2%
    - data cache miss rate 4%
    - miss penalty always 40 cycles
    - CPI of 2 without memory stalls
    - frequency of load/stores 36% of instructions
1. *How much faster is a machine with a perfect cache that never misses?*
  2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*
  3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*

# Solution

1.

- Assume instruction count =  $I$
- Instruction miss cycles =  $I \times 2\% \times 40 = 0.8 \times I$
- Data miss cycles =  $I \times 36\% \times 4\% \times 40 = 0.576 \times I$
- So, total memory-stall cycles =  $0.8 \times I + 0.576 \times I = 1.376 \times I$ 
  - in other words, 1.376 stall cycles per instruction
- Therefore, CPI with memory stalls =  $2 + 1.376 = 3.376$
- Assuming instruction count and clock rate remain same for a perfect cache and a cache that misses:  
CPU time with stalls / CPU time with perfect cache  
 $= 3.376 / 2 = 1.688$
- Performance with a perfect cache is better by a factor of 1.688



## Solution (cont.)

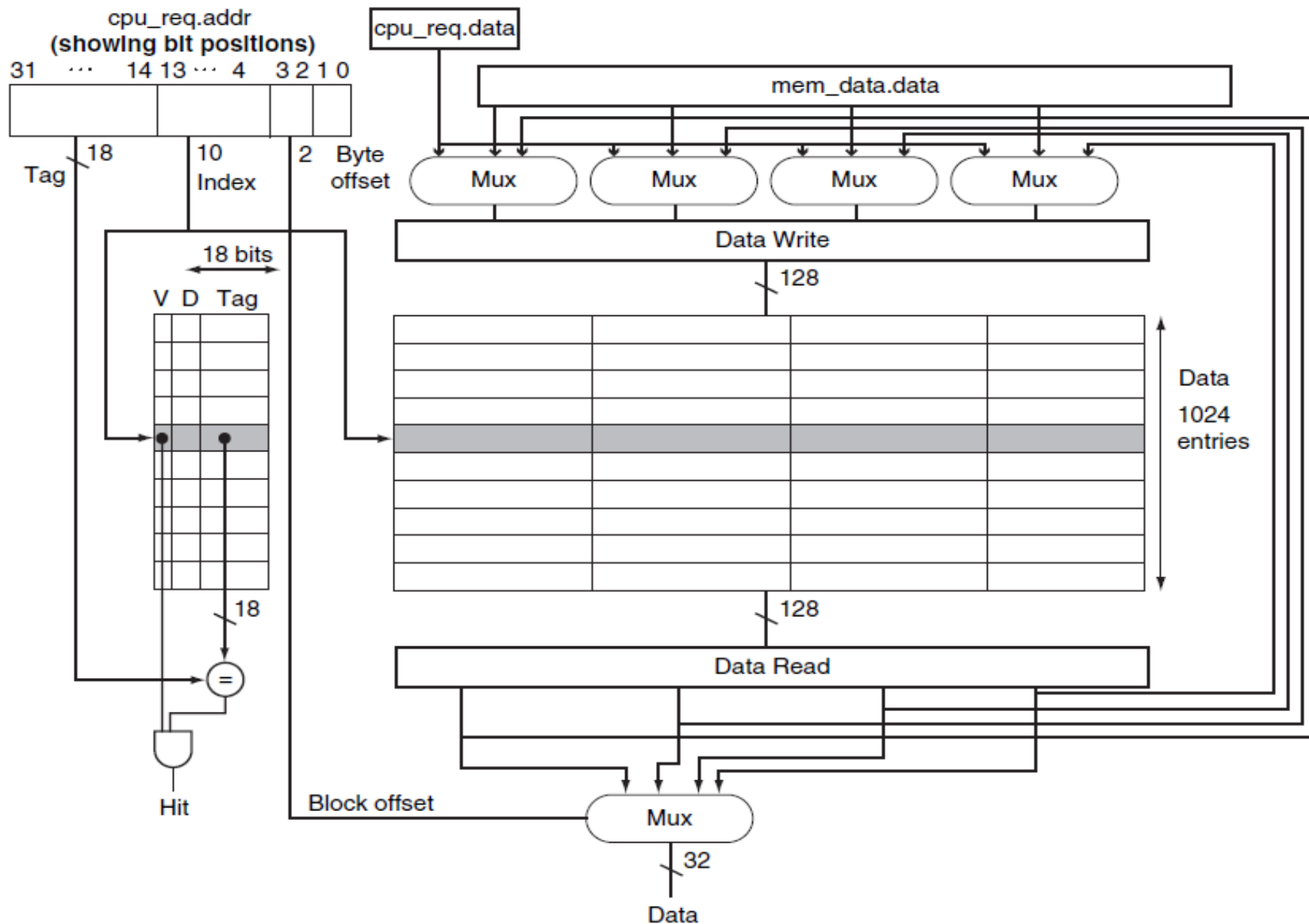
2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*

- CPI without stall = 1
- CPI with stall =  $1 + 1.376 = 2.376$  (clock has not changed so stall cycles per instruction remains same)
- CPU time with stalls / CPU time with perfect cache  
= CPI with stall / CPI without stall  
= 2.376
- Performance with a perfect cache is better by a factor of 2.376
- **Conclusion:** Lower the CPI more pronounced is the impact of stall cycles

## Solution (cont.)

3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*

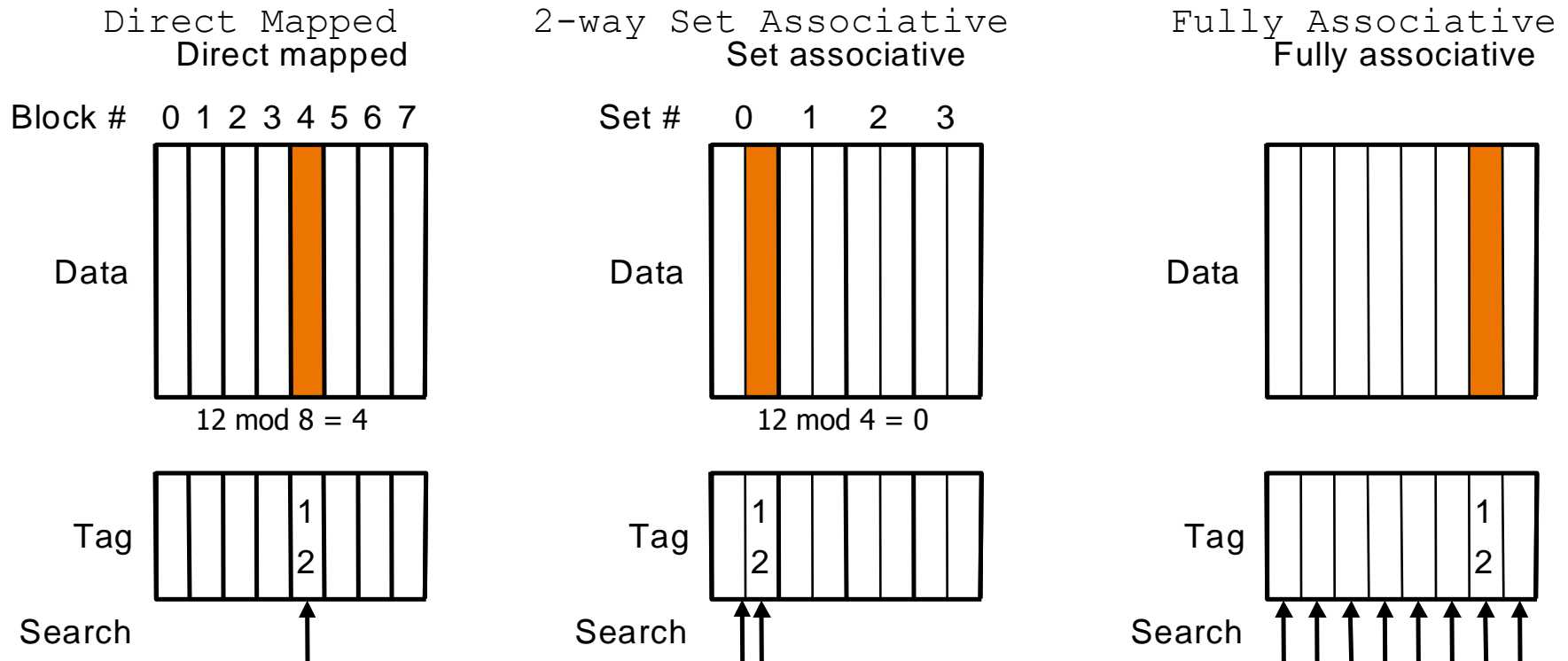
- With doubled clock rate, miss penalty =  $2 \times 40 = 80$  clock cycles
- Stall cycles per instruction =  $(I \times 2\% \times 80) + (I \times 36\% \times 4\% \times 80)$   
 $= 2.752 \times I$
- So, faster machine with cache miss has CPI =  $2 + 2.752 = 4.752$
- CPU time with stalls / CPU time with perfect cache  
= CPI with stall / CPI without stall  
=  $4.752 / 2 = 2.376$
- Performance with a perfect cache is better by a factor of 2.376
- **Conclusion:** with higher clock rate cache misses “hurt more” than with lower clock rate



# Decreasing Miss Rates with Associative Block Placement

- *Direct mapped*: one *unique* cache location for each memory block
  - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
  - *all* cache entries are searched (in parallel) to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size *n* it is *n*-way set-associative
  - cache set address = memory block address *mod* number of sets in cache
  - all cache entries in the corresponding set are searched (*in parallel*) to locate block
- Increasing degree of associativity
  - *reduces miss rate*
  - *increases hit time* because of the parallel search and then fetch

# Decreasing Miss Rates with Associative Block Placement



**Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity**

# Decreasing Miss Rates with Associative Block Placement

One-way set associative  
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

**Configurations of an 8-block cache with different degrees of associativity**

# Example

- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*

*0, 8, 0, 6, 8,*

*for each of the following cache configurations*

- 1. direct mapped*
- 2. 2-way set associative (use LRU replacement policy)*
- 3. fully associative*

- Note about LRU replacement
  - in a 2-way set associative cache LRU replacement can be implemented with one bit at each set whose value indicates the most recently referenced block

# Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 ( $= 0 \bmod 4$ )
6	2 ( $= 6 \bmod 4$ )
8	0 ( $= 8 \bmod 4$ )

## Block address translation in direct-mapped cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – **red** indicates new entry added

- 5 misses



# Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 ( $= 0 \bmod 2$ )
6	0 ( $= 6 \bmod 2$ )
8	0 ( $= 8 \bmod 2$ )

## Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – **red** indicates new entry added

- Four misses

# Solution (cont.)

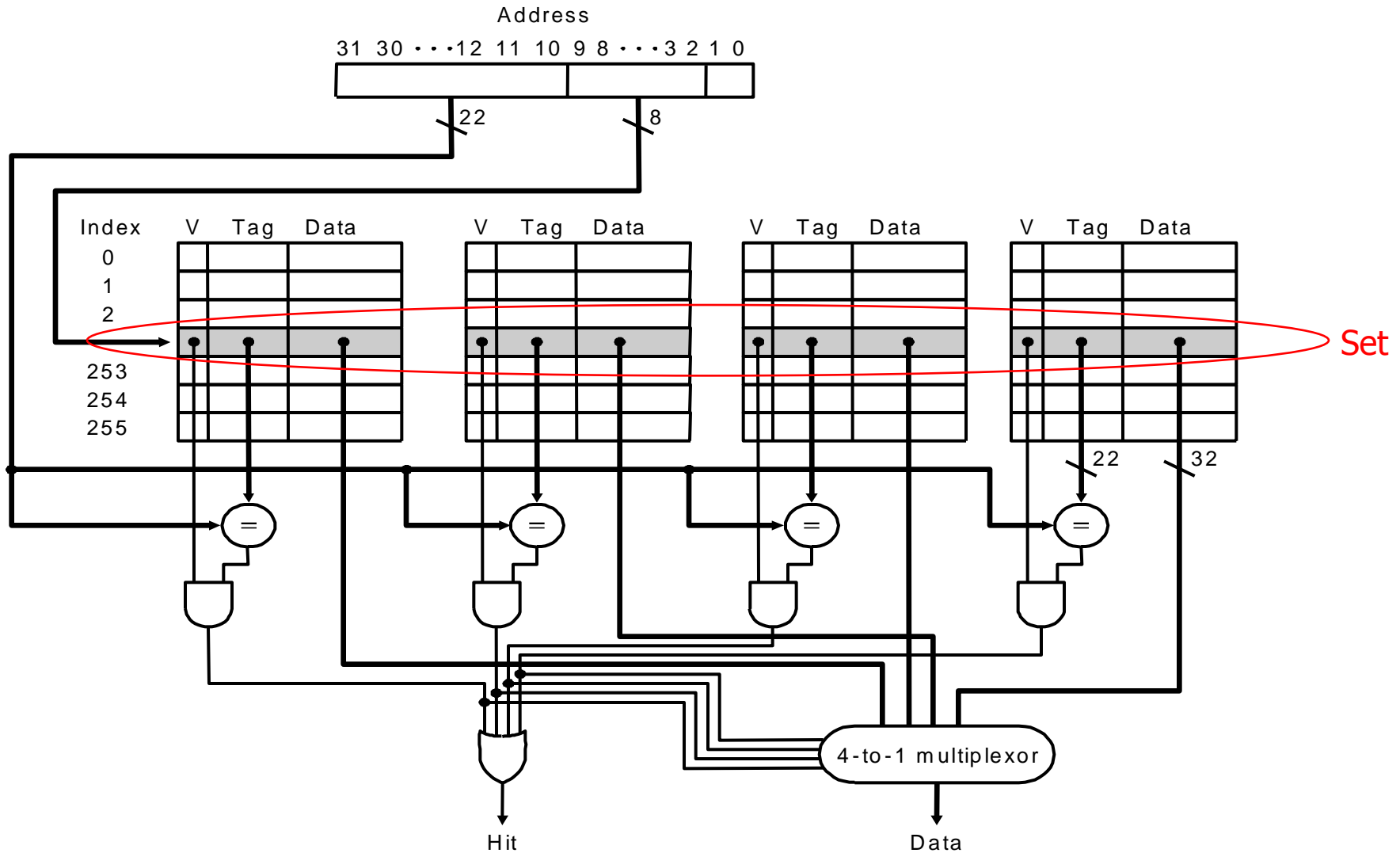
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

**Cache contents after each reference – red indicates new entry added**

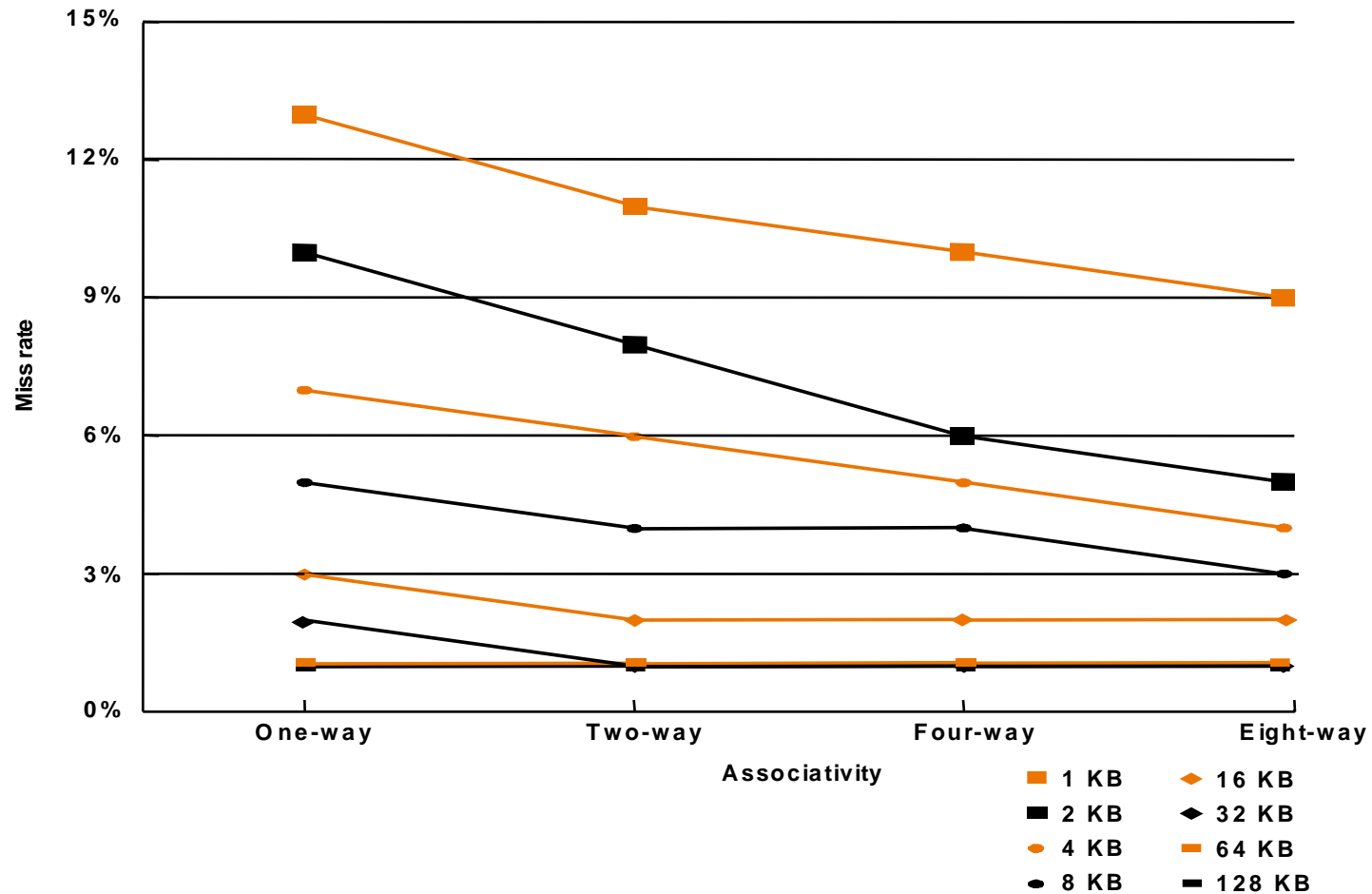
- 3 misses

# Implementation of a Set-Associative Cache



**4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor: size of cache is 1K blocks = 256 sets \* 4-block set size**

# Performance with Set-Associative Caches



**Miss rates for each of eight cache sizes  
with increasing associativity:  
data generated from SPEC92 benchmarks  
with 32 byte block size for all caches**



# Replacement Policy

---

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity



# Multilevel Caches

---

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

# Decreasing Miss Penalty with Multilevel Caches

- Add a *second-level* cache
  - primary cache is on the same chip as the processor
  - use SRAMs to add a second-level cache, *between main memory and the first-level cache*
  - if miss occurs in primary cache second-level cache is accessed
  - if data is found in second-level cache miss penalty is access time of second-level cache which is much less than main memory access time
  - if miss occurs again at second-level then main memory access is required and large miss penalty is incurred
- Design considerations using two levels of caches:
  - try and optimize the *hit time on the 1<sup>st</sup> level cache* to reduce clock cycle
  - try and optimize the *miss rate on the 2<sup>nd</sup> level cache* to reduce memory access penalties
  - In other words, 2<sup>nd</sup> level allows 1<sup>st</sup> level to go for speed without “worrying” about failure...

## Example Problem

- Assume a 500 MHz machine with
  - base CPI 1.0
  - main memory access time 200 ns.
  - miss rate 5%
- *How much faster will the machine be if we add a second-level cache with 20ns access time that decreases the miss rate to 2%?*

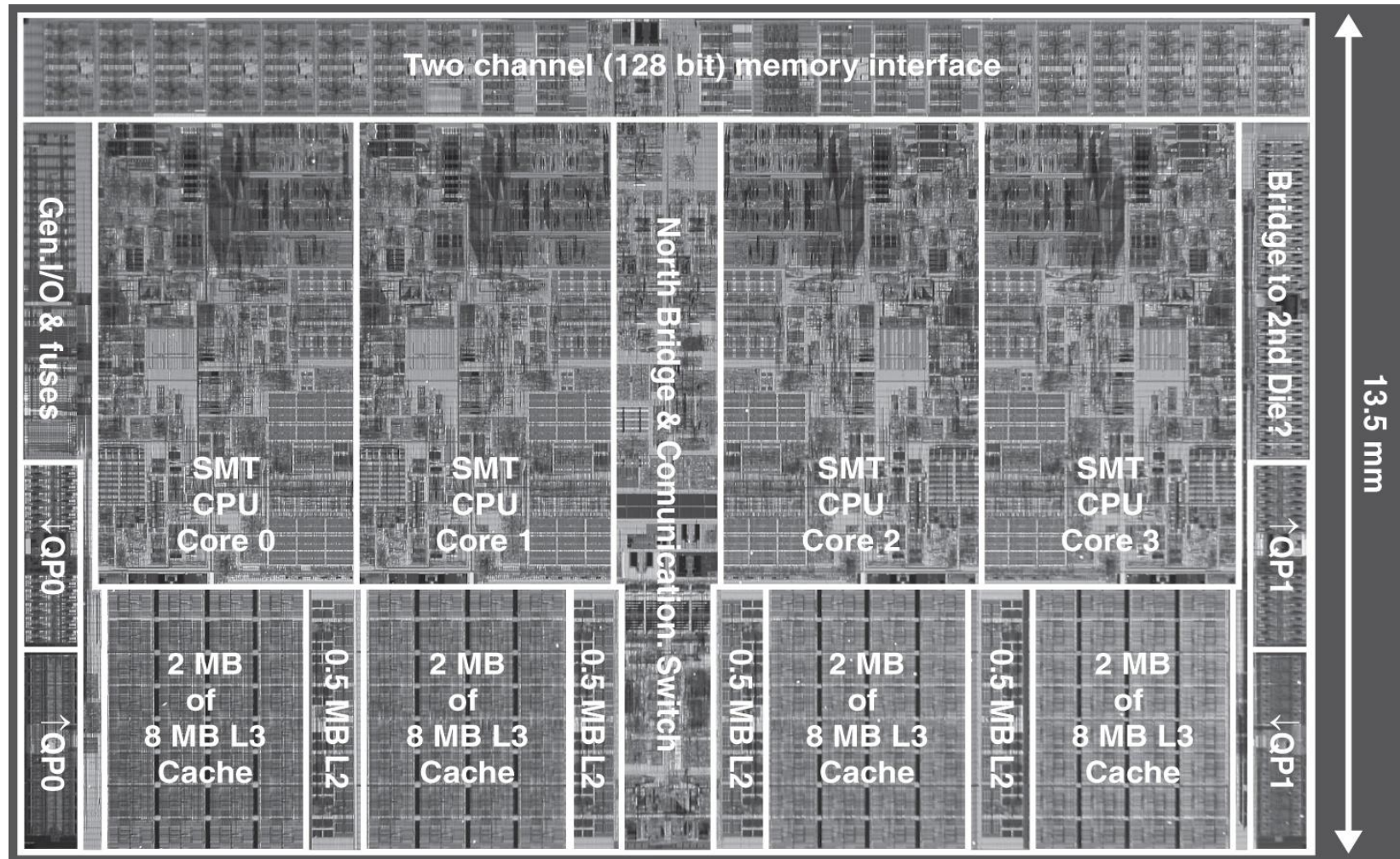


# Solution

- Miss penalty to main =  $200 \text{ ns} / (2 \text{ ns} / \text{clock cycle}) = 100 \text{ clock cycles}$
- Effective CPI with one level of cache  
= Base CPI + Memory-stall cycles per instruction  
=  $1.0 + 5\% \times 100 = 6.0$
- With two levels of cache, miss penalty to second-level cache  
=  $20 \text{ ns} / (2 \text{ ns} / \text{clock cycle}) = 10 \text{ clock cycles}$
- Effective CPI with two levels of cache  
= Base CPI + Primary stalls per instruction  
+ Secondary stall per instruction  
=  $1 + 5\% \times 10 + 2\% \times 100 = 3.5$   
=  $1 + (5\% - 2\%) \times 10 + 2\% \times (100 + 10)$
- Therefore, machine with secondary cache is faster by a factor of  
 $6.0 / 3.5 = 1.71$

# Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

# 3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	512KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available