L8 Cache Part I

Zonghua Gu, 2018

Acknowledgement: some slides taken from UC Berkeley CS61C

Outline

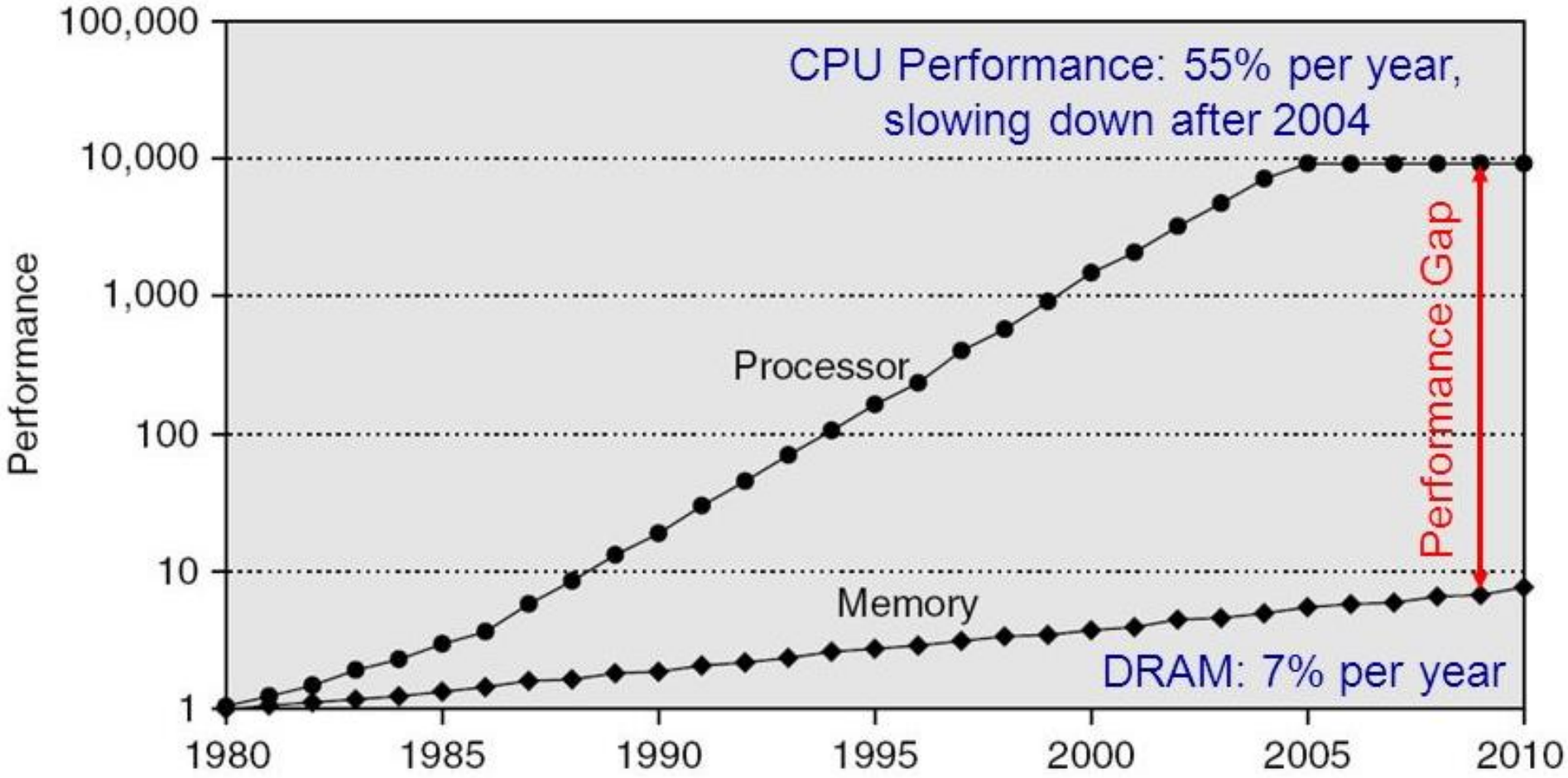
- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

Why are Large Memories Slow? Library Analogy

- Time to find a book in a large library
  - Search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book
- Larger libraries worsen both delays
- Electronic memories have same issue, *plus* the technologies used to store a bit slow down as density increases (e.g., SRAM vs. DRAM vs. Disk)

However what we want is a large yet fast memory!

Processor-DRAM Gap (Latency)



1980 microprocessor executes **~one instruction** in same time as DRAM access

2017 microprocessor executes **~1000 instructions** in same time as DRAM access

Memory wall: memory access likely to be the performance bottleneck

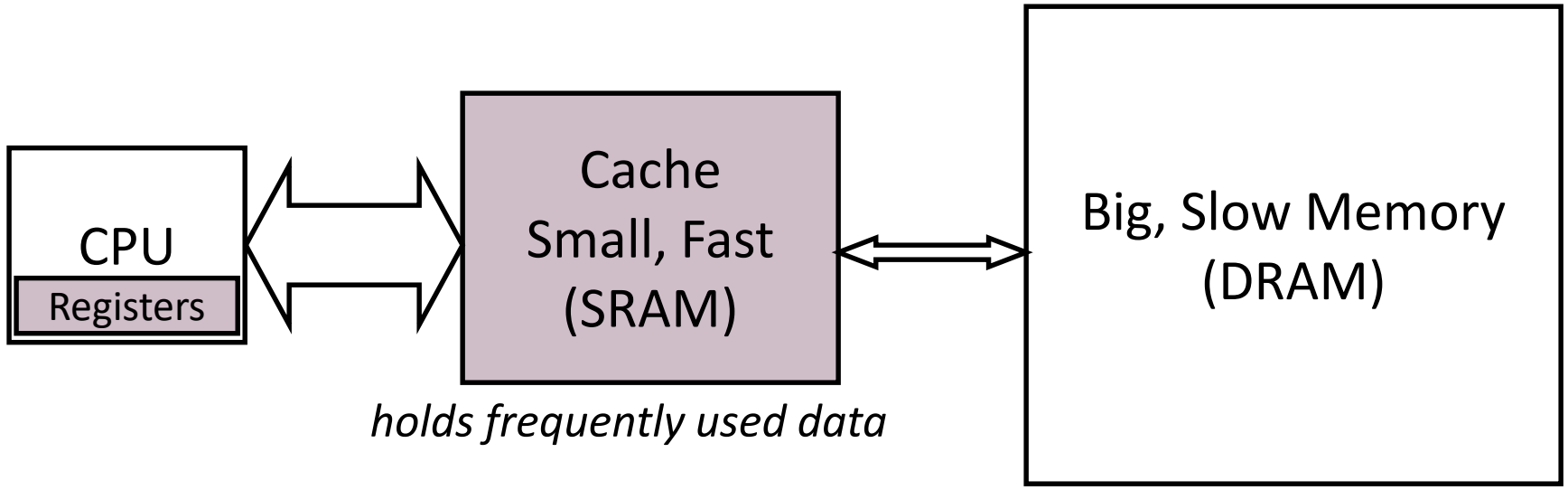
What To Do: Library Analogy

- Write a report using library books
- Go to library, look up relevant books, fetch from stacks, and place on desk in library
- If need more, check them out and keep on desk
  - But don't return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only 0.00001% of all books in the library

Outline

- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

Memory Hierarchy



- *Capacity*: register << cache (typically on-chip) << memory (off-chip)
- *Latency*: register << cache (typically on-chip) << memory (off-chip)

On a data access:

if data $\in$ cache $\Rightarrow$ cache hit $\Rightarrow$ low latency access *(SRAM)*

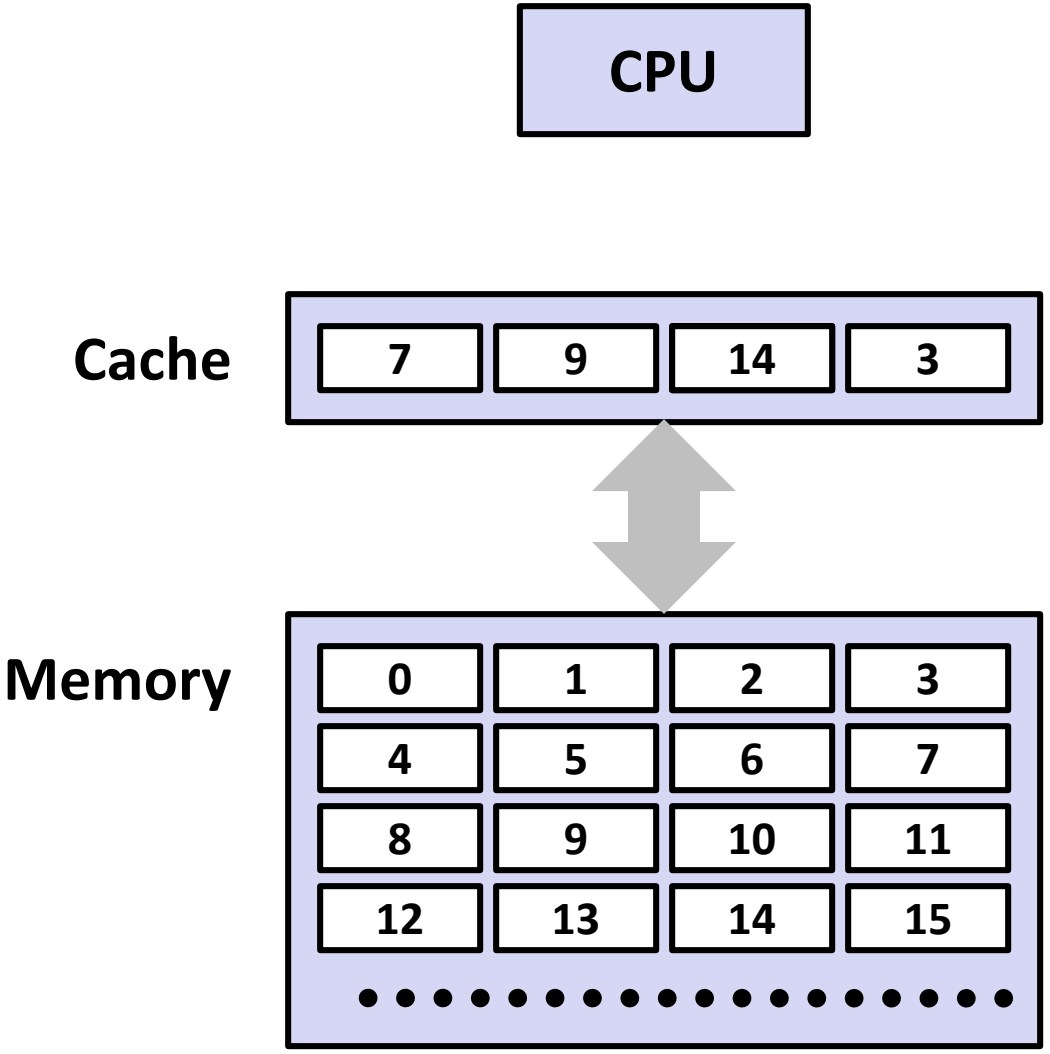
if data $\notin$ cache $\Rightarrow$ cache miss $\Rightarrow$ high latency access *(DRAM)*

Goal: create the illusion of accessing as much memory as is available in the slow memory at the speed of the fast cache

Memory Hierarchy Technologies

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

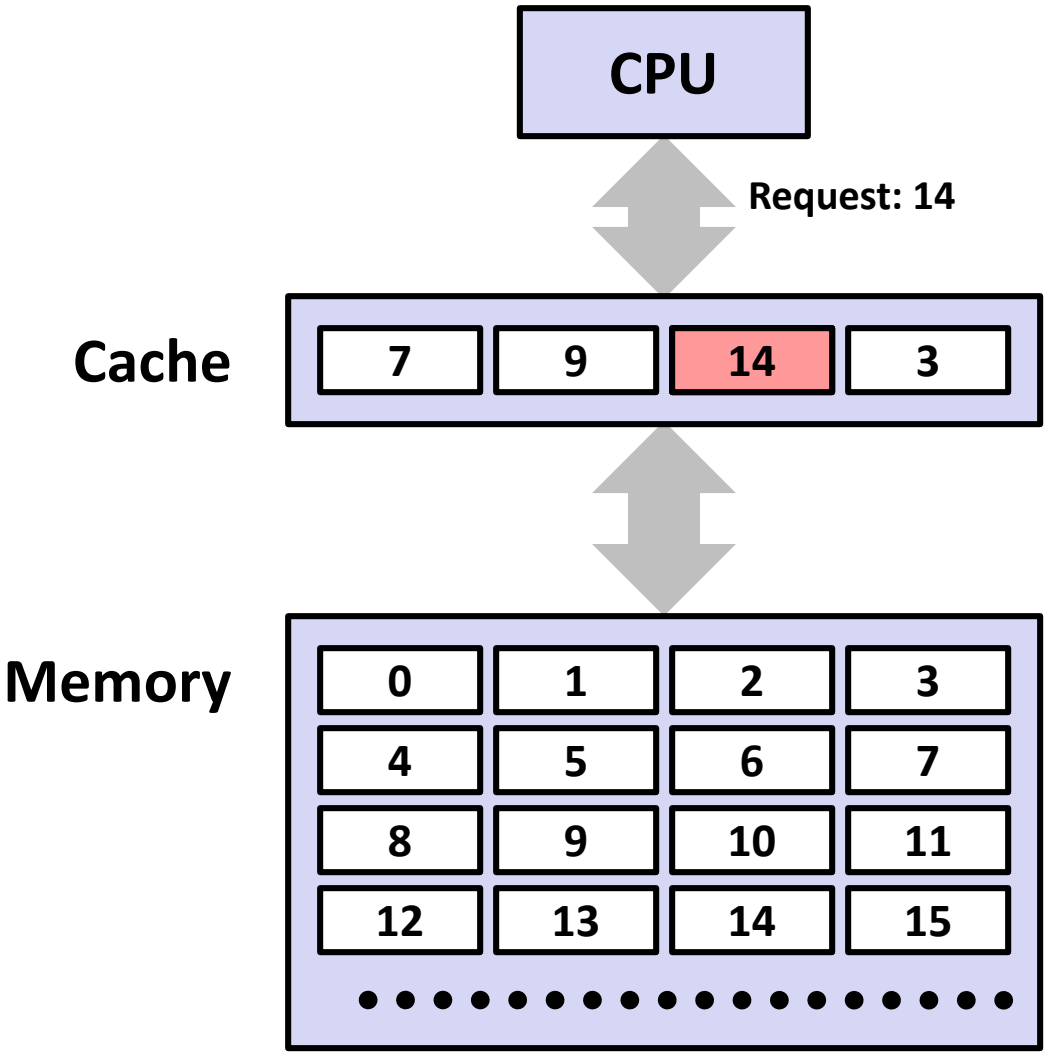
Cache Blocks



- Smaller, faster, more expensive memory
- Typical block size: 1 – 4 words (4 – 16 Bytes)

- Larger, slower, cheaper memory.

General Cache Concepts: Hit

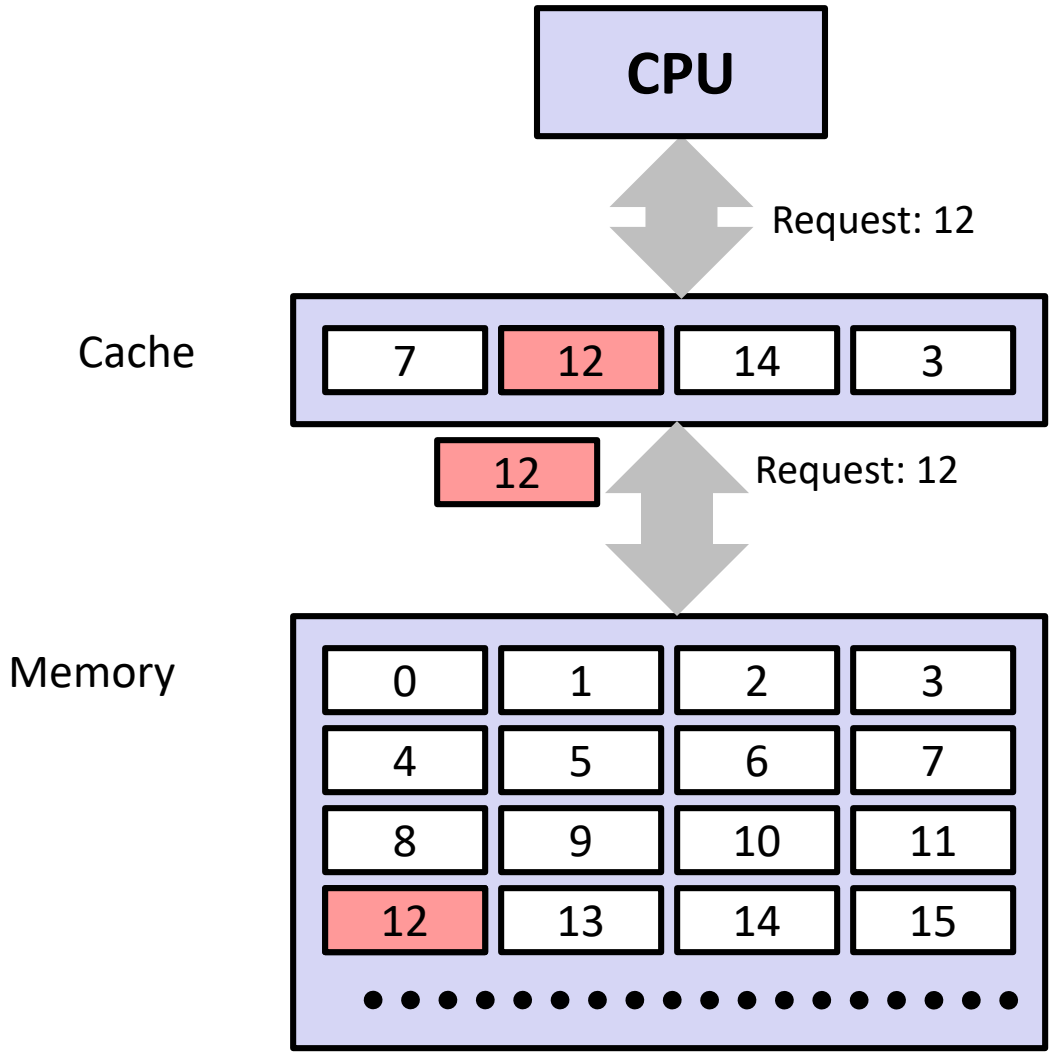


Data in block b is needed

Block b is in cache:
Hit!

Data is loaded from cache into CPU register

General Cache Concepts: Miss



Data in block b is needed

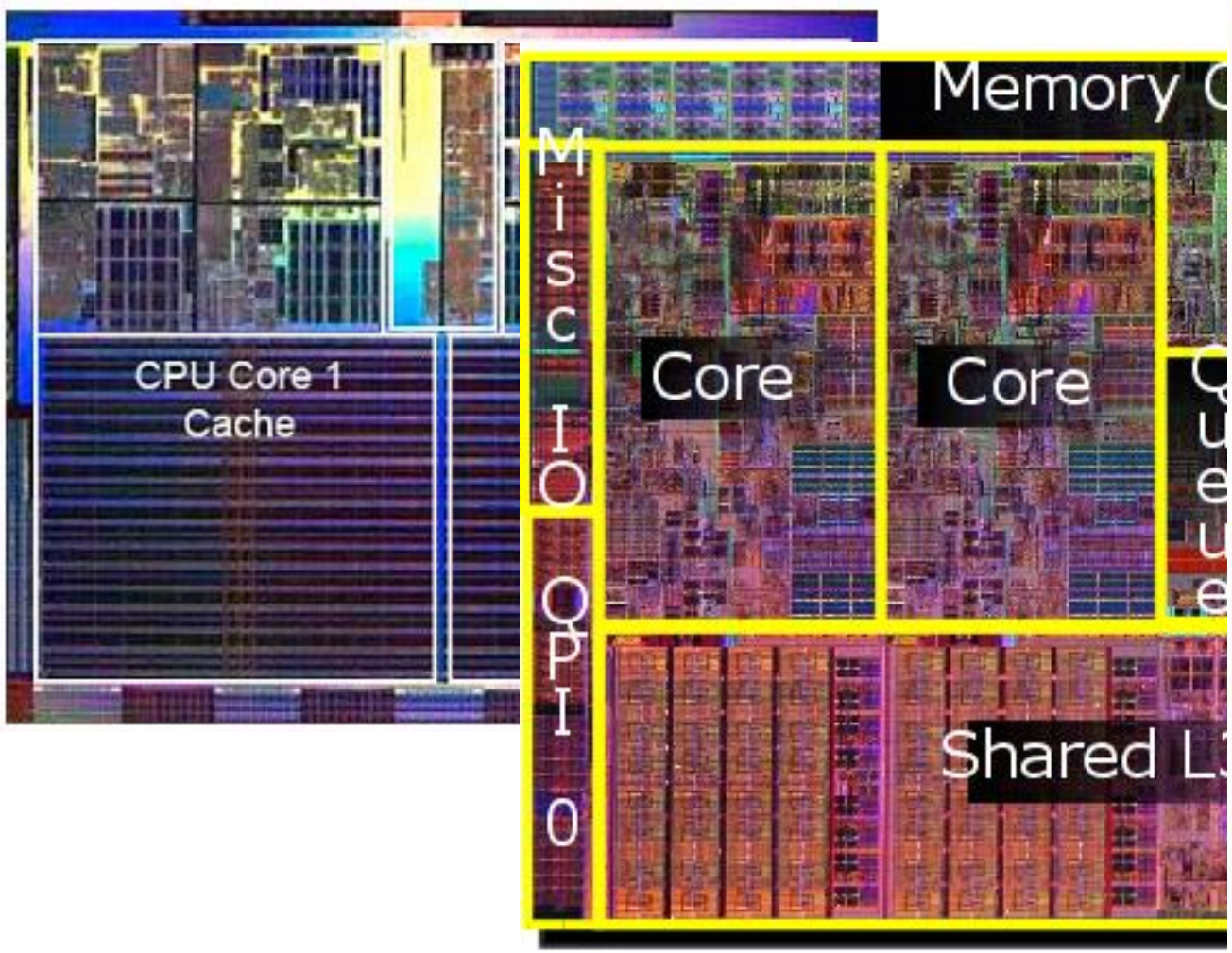
Block b is not in cache:
Miss!

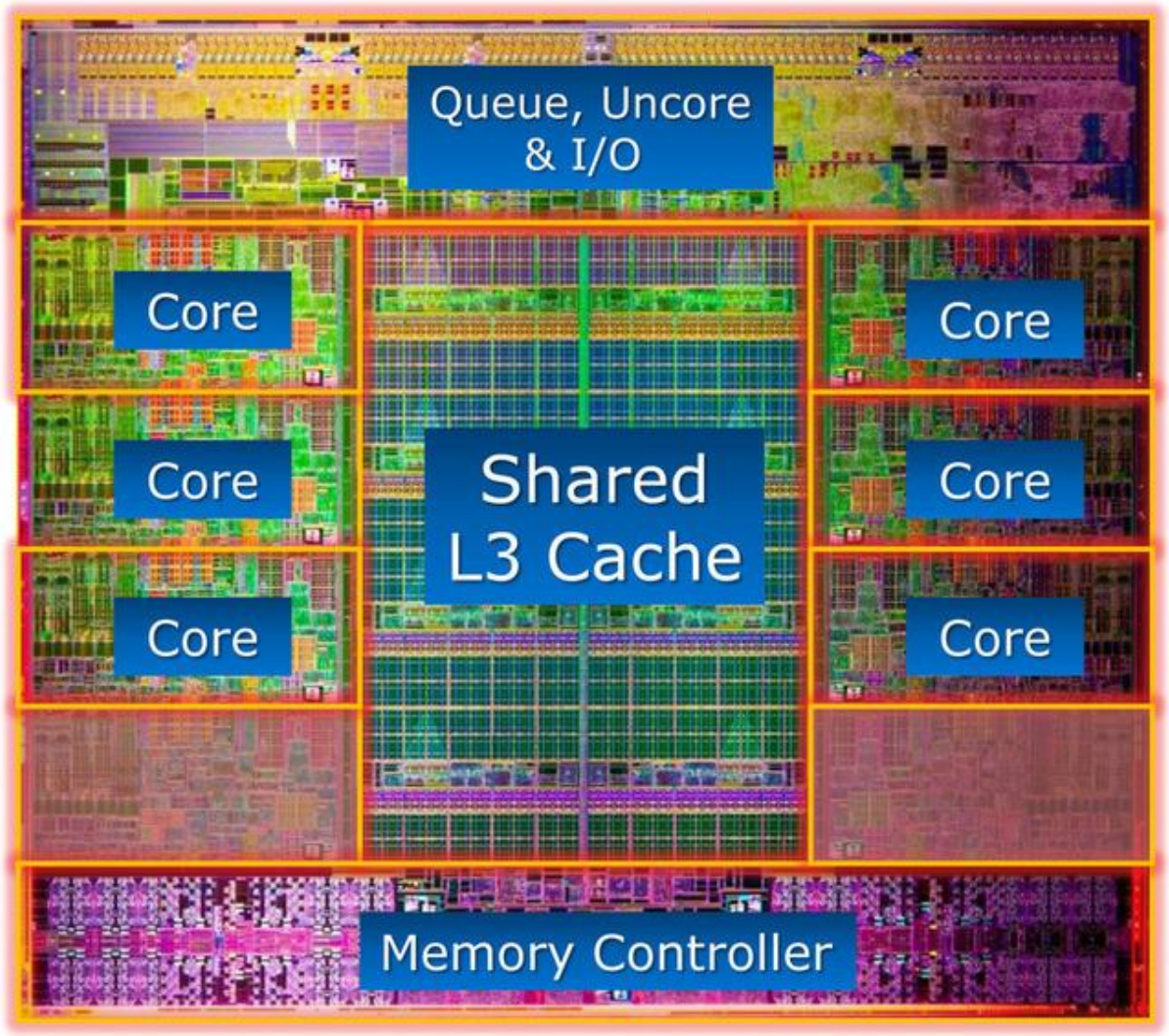
Block b is fetched from *memory*

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

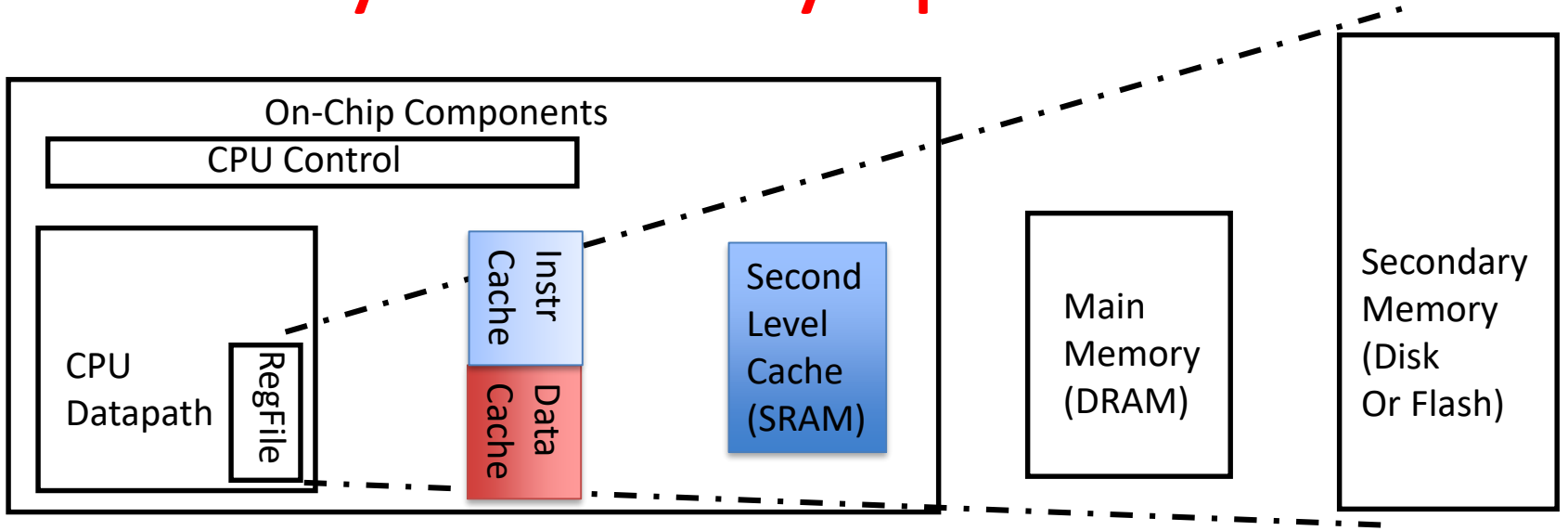
Data is loaded from cache into CPU register

Chip Photos


CPU Core 1
Cache
Memory C
Misc IO
QPI 0
Core
Core
Shared L


Queue, Uncore
& I/O
Core
Core
Core
Core
Core
Core
Shared
L3 Cache
Memory Controller

Memory Hierarchy Speed vs. Size



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100B's	10KB's	MB's	GB's	TB's
Cost/bit:	highest	←		→	lowest

How is the Hierarchy Managed?

- Registers ↔ memory hierarchy
  - By compiler (or assembler programmer)
- Cache ↔ main memory
  - By the cache controller hardware
  - Focus of this lecture
- Main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
  - By the programmer (files)

Principle of Locality

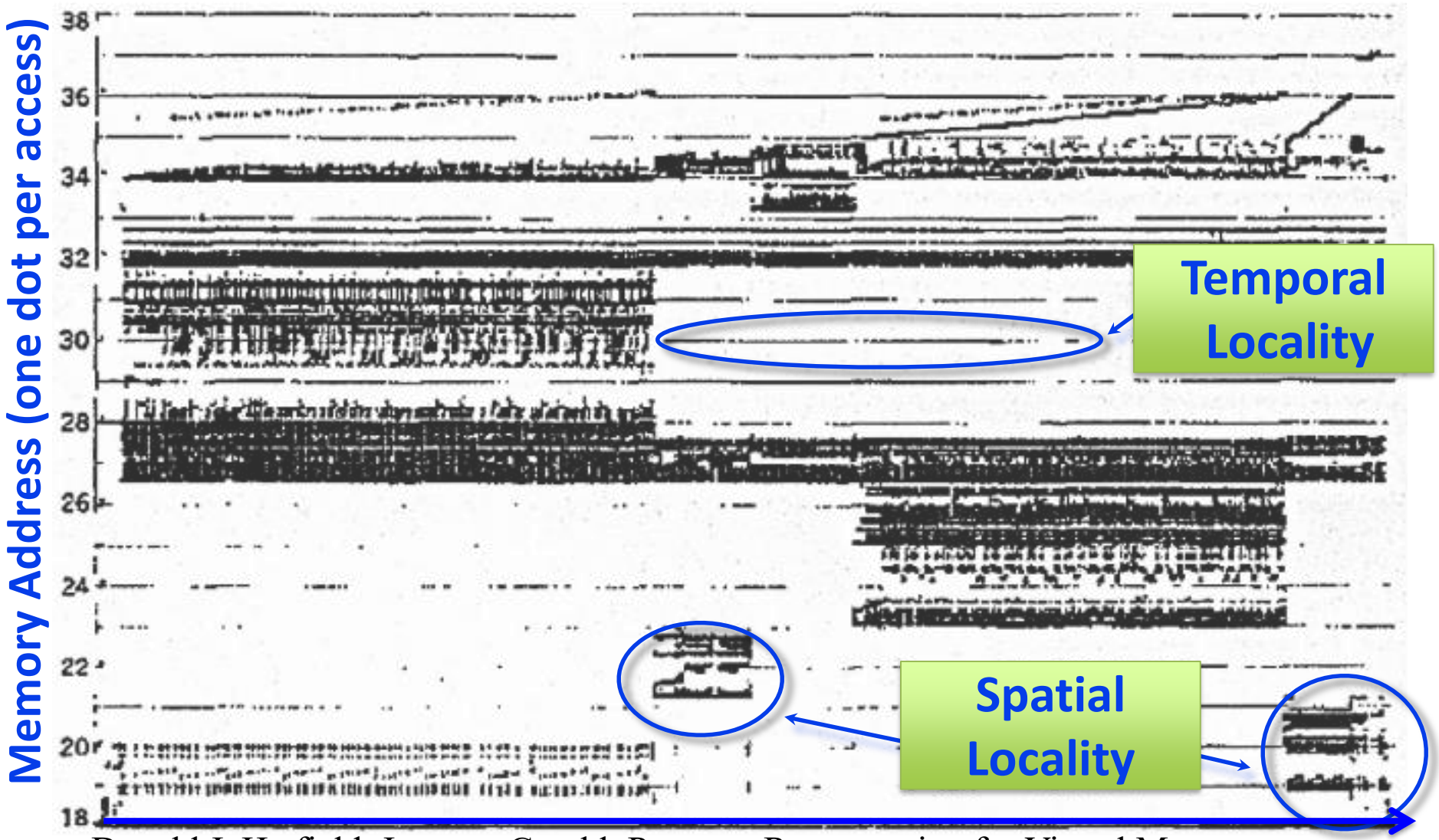
- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- *Temporal Locality* (locality in time)
  - Go back to same book on desk multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
    - Keep recently-accessed blocks in the cache
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books around the book you want, since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
    - When fetching a block into cache, also fetch blocks around it

Quiz: What locality does this program have?

```
int sum = 0, a[n];
…
for (i = 0; i < n; i++) {
  sum += a[i];
}
return sum;
```

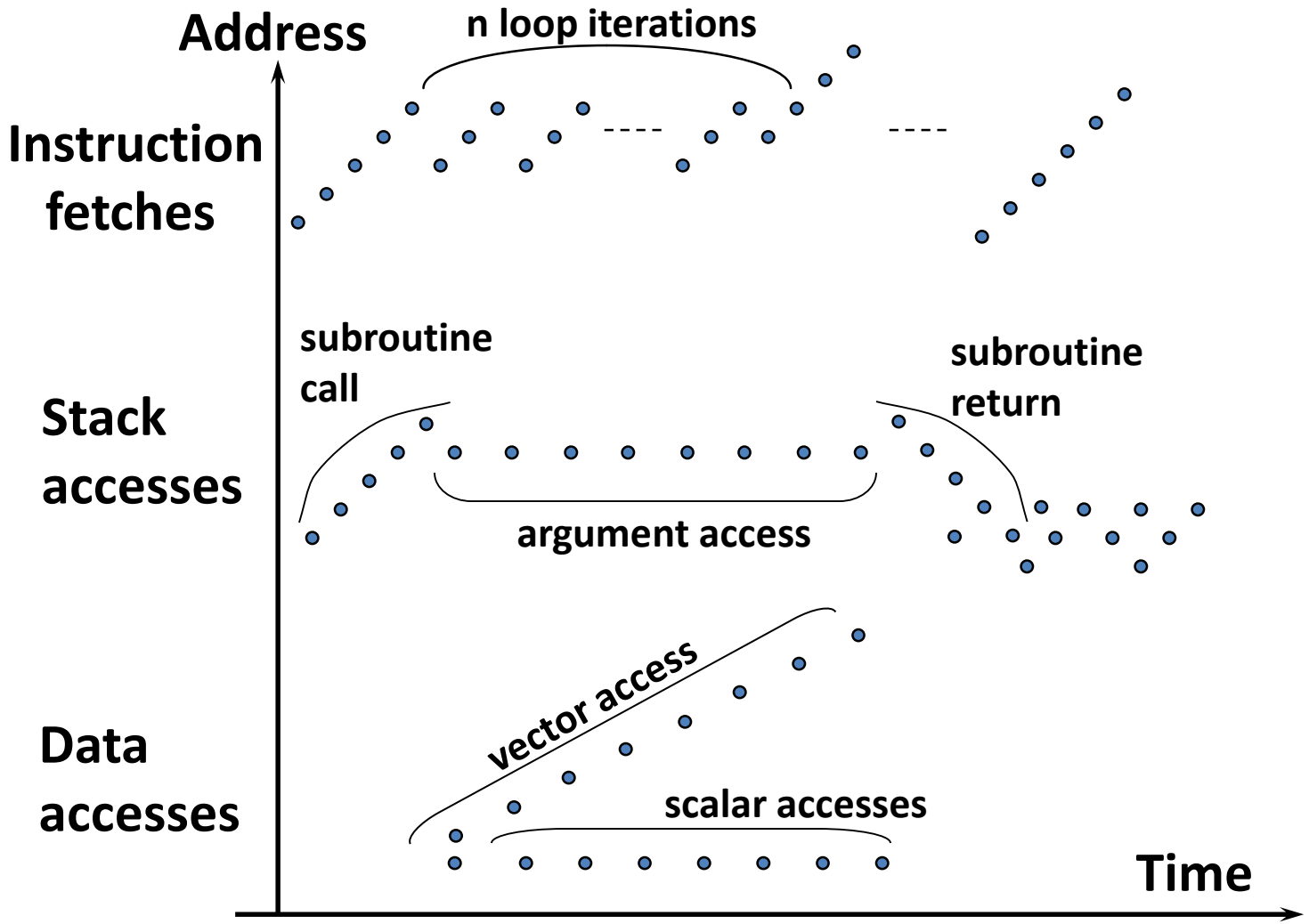
- Data:
  - Temporal locality: variable sum is referenced in each iteration
  - Spatial locality: array a[] is accessed with stride-1 in each iteration (assuming a[] is stored in contiguous addresses in memory)
- Instructions:
  - Temporal locality: the loop body is executed repeatedly for n times
  - Spatial locality: instructions are accessed sequentially (with 1 branch in each iteration) (assuming instructions are stored in contiguous addresses in memory)

Memory Reference Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

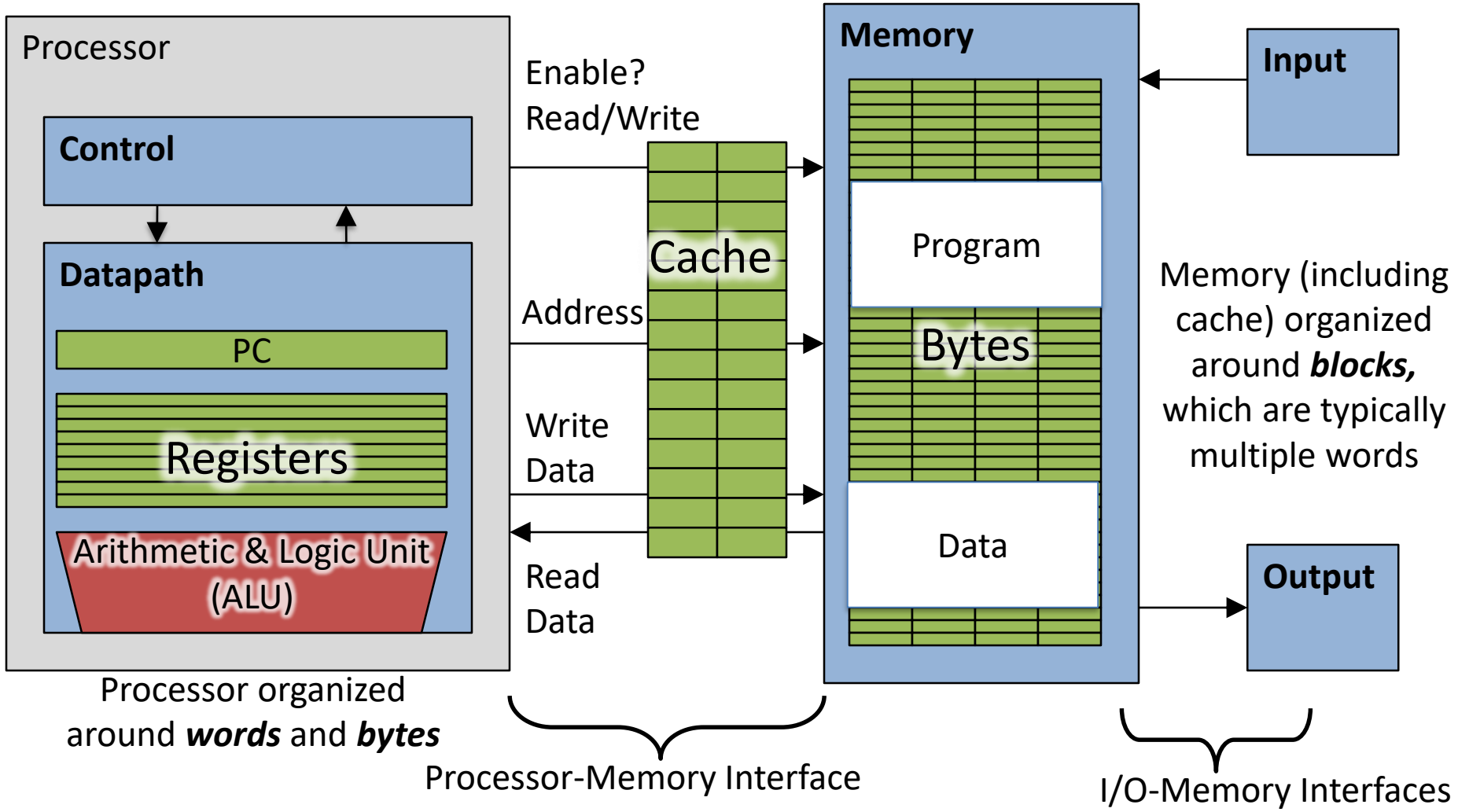
Memory Reference Patterns



Outline

- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

Processor with Cache



Cache vs. Memory

- Cache size << memory size
  - Smaller cache is faster
- 1-to-many correspondence between cache blocks and memory blocks
  - Use *Tags* in the cache to match cache and memory blocks
- A cache block is also called a cache line

Block Must be Aligned in Memory

- If each cache block is 4 Bytes (1 word), then binary address each cache block always ends in 00_{two}
- How to take advantage of this to save hardware and energy?
- Don't need to compare last 2 bits of 32-bit byte address (comparator can be narrower)
  - Don't need to store last 2 bits of 32-bit byte address in Cache Tag (Tag can be narrower)
- If each cache block is 8 Bytes (2 words), then binary address each cache block always ends in 000_{two}

Tradeoffs of Cache Block Sizes

- Smaller cache blocks → more fine-grained caching → take better advantage of temporal locality
  - A given data item stays in the cache longer before getting replaced
- Larger cache blocks → more coarse-grained caching, take better advantage of spatial locality
  - Fetching each cache block brings in lots of data at nearby addresses into the cache
- In either case, if the program has poor temporal or spatial locality, then lots of junk may be brought into cache

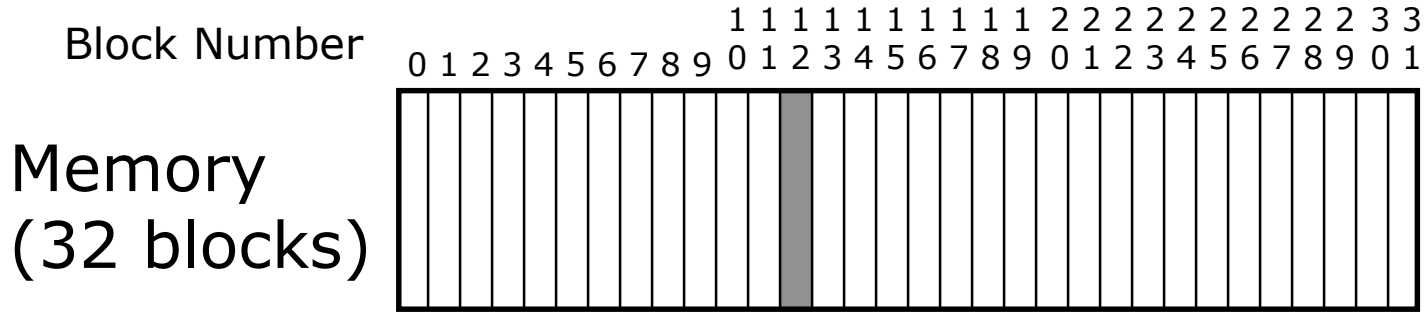
Outline

- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

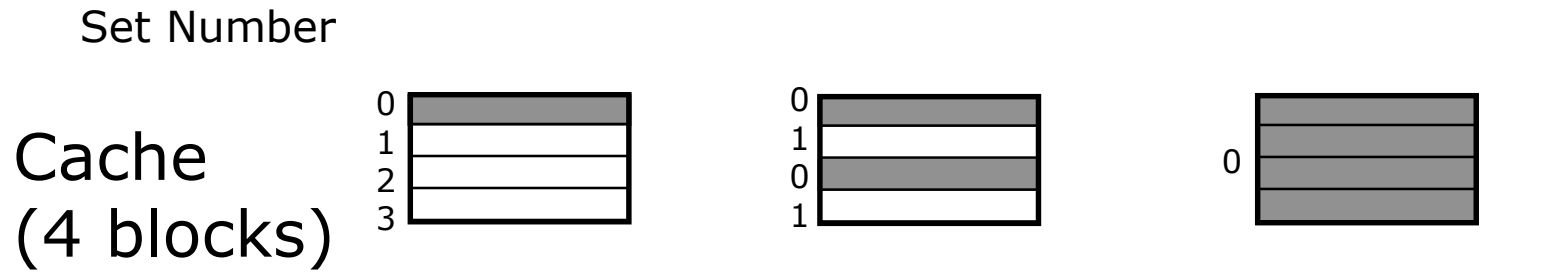
Alternative Cache Organizations

- A memory block is mapped to one cache set, which may contain one or more cache blocks
- Direct Mapped (DM)
  - Each cache set has 1 cache block; # cache sets = # cache blocks
  - A memory block is mapped to 1 possible cache block
- Fully Associative (FA)
  - A single cache set contains all cache blocks; # cache sets = 1
  - A memory block can be mapped to any cache block
- N-way Set Associative (SA)
  - Each cache set has N cache blocks; # cache sets = # cache blocks / N
  - N is also called associativity
  - A memory block can be mapped to one of N possible cache blocks
- DM and FA are special cases of SA
  - DM = 1-way SA (N = 1)
  - FA = N-way SA (N = total number of cache blocks)

Alternative Cache Organizations (4-block cache)

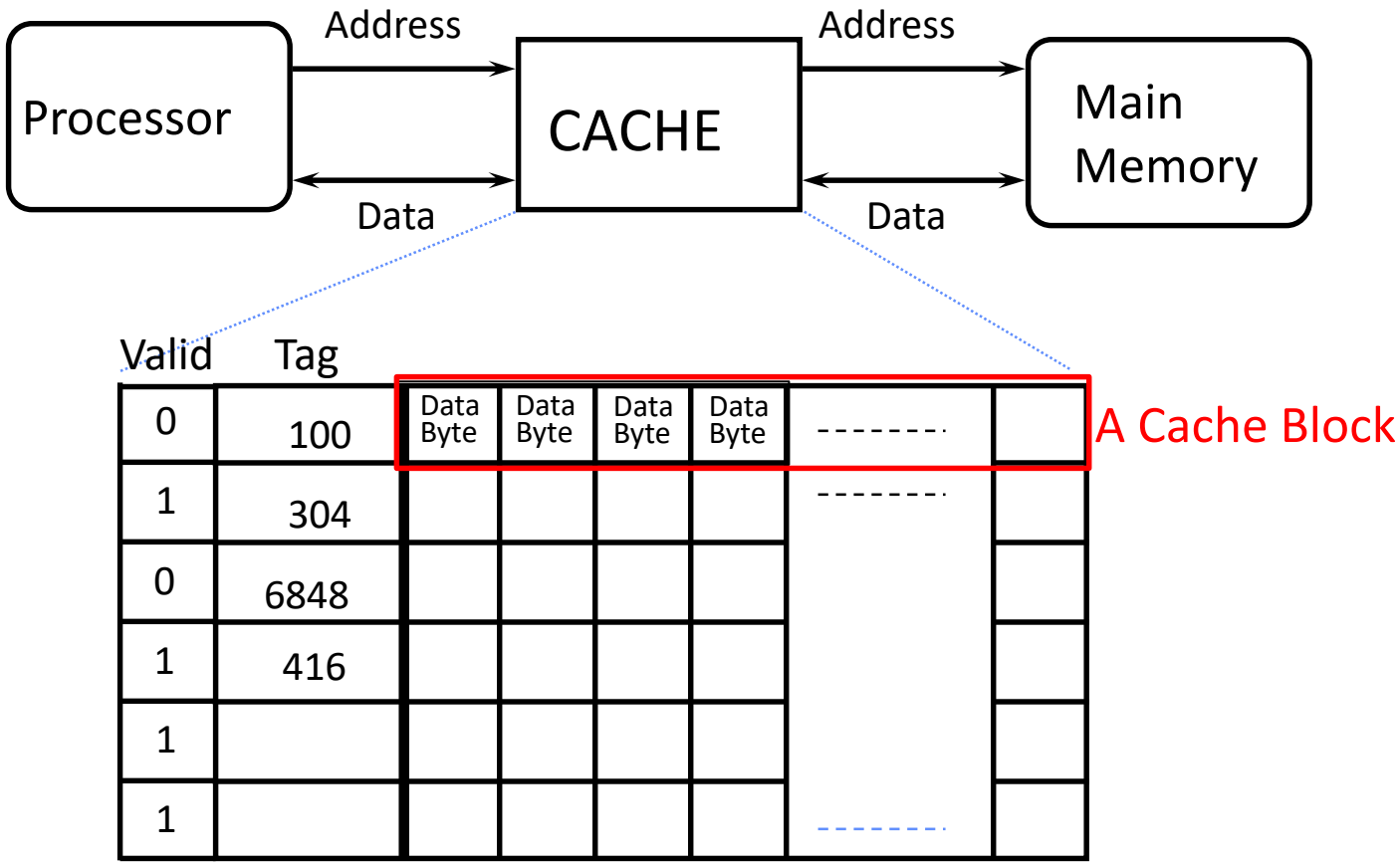


Where are possible locations in cache that block #12 in memory can be placed?



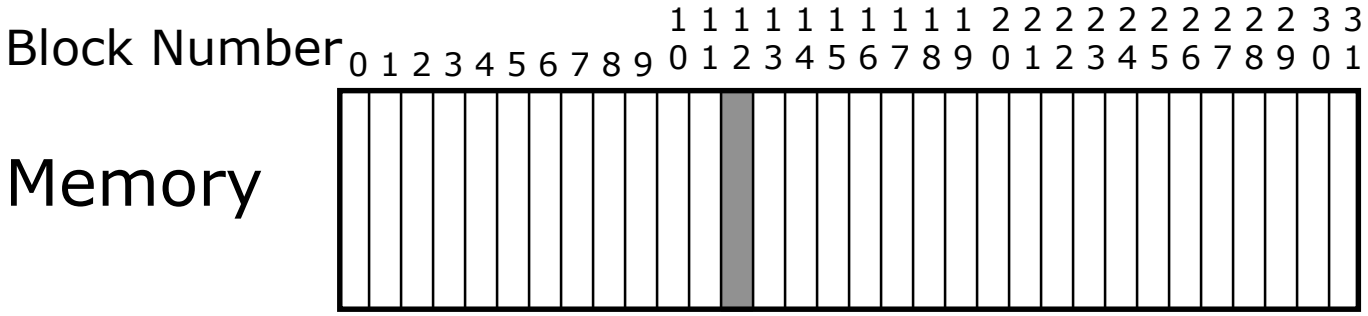
DM	2-way SA	FA (4-way SA)
In set 0 (1 block)	In set 0 (2 blocks)	In set 0 (4 blocks)

Inside a Cache

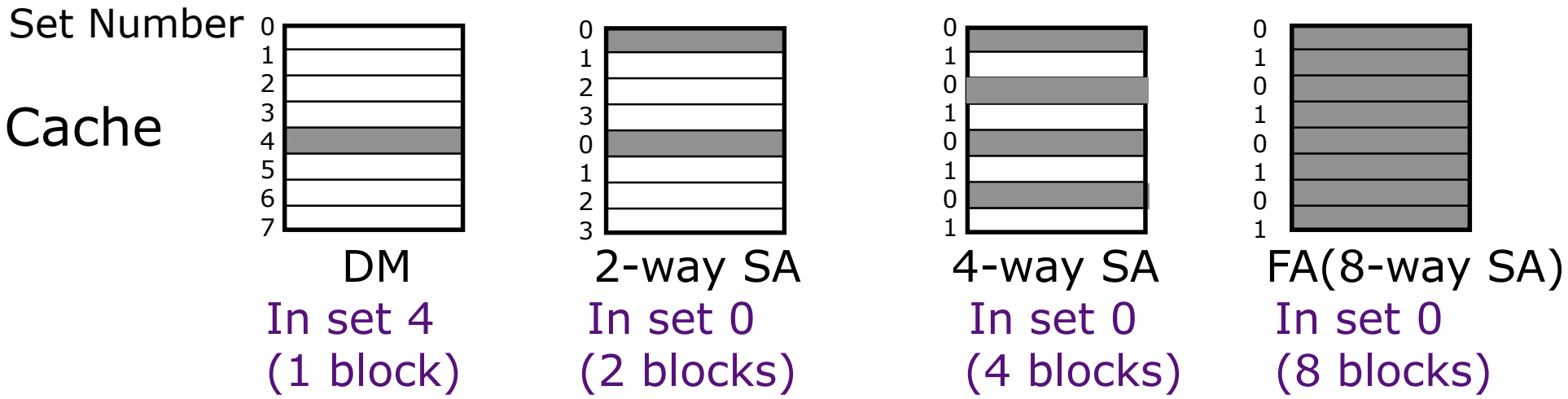


- A "valid bit" indicates of a cache block contains valid data
  - e.g., upon startup, the cache is "cold": all cache blocks are invalid
  - The cache is "warmed-up" gradually by bringing content into the cache
- A tag helps identify the memory block contained in the cache block;
  - Disambiguate among multiple possible memory blocks that may be mapped to the same cache block
- Cache capacity refers to the total size of cache blocks (not including Tag and Valid bits)

Alternative Cache Organizations (8-block cache)

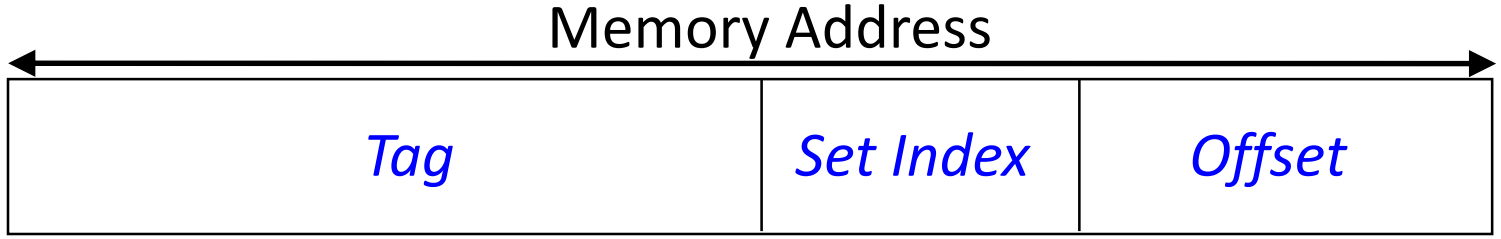


Where are possible locations in cache that block #12 in memory can be placed?



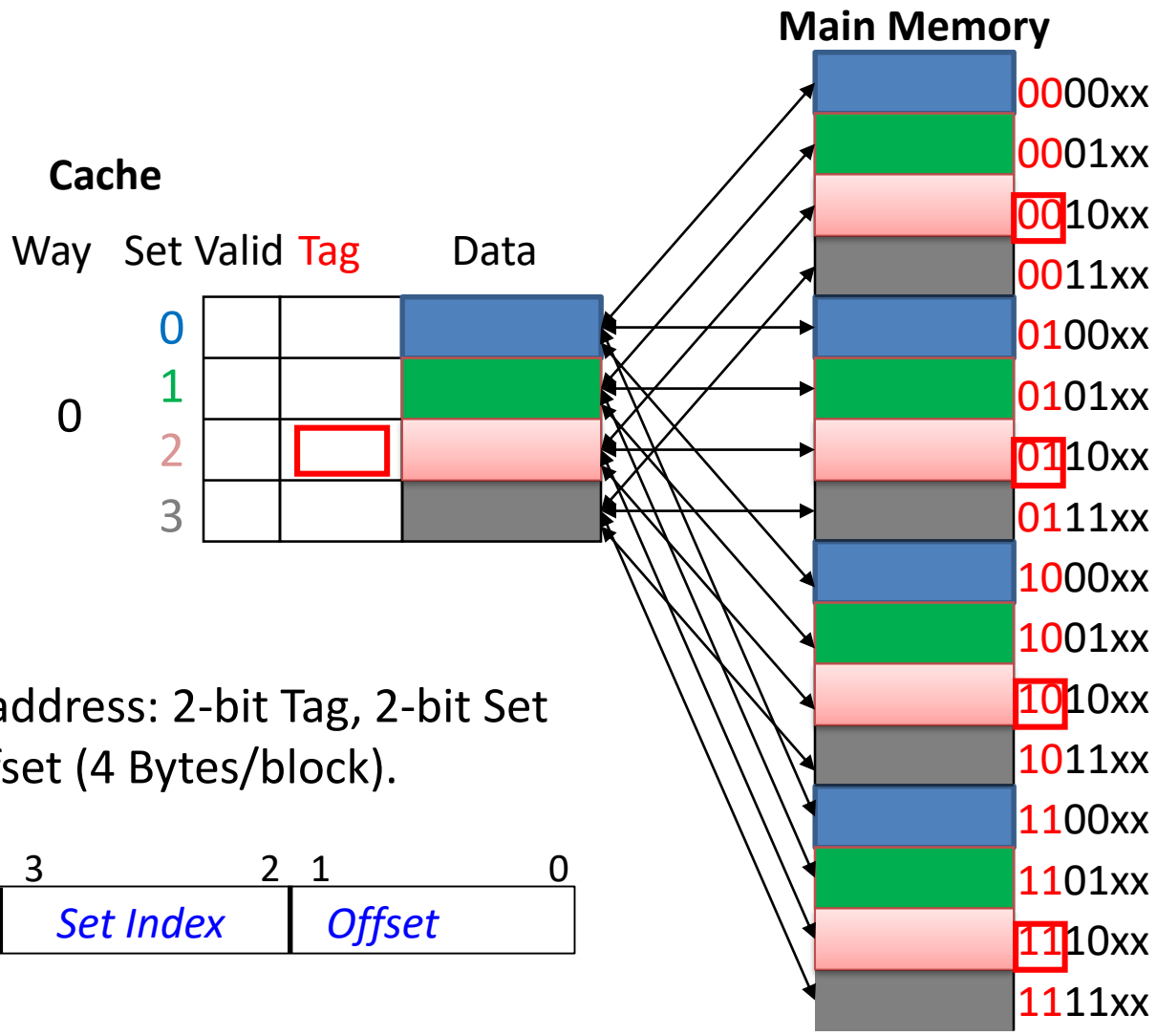
Processor Address Fields Used by Cache Controller

- Offset: Byte address within a cache block
- Set Index: Selects which set
- Tag: Remaining portion of processor address



- Size of Set Index (SI) = $\log_2$(number of sets)
- Size of Offset = $\log_2$(number of bytes/block)
- Size of Tag = Address size – Size of SI - Size of Offset

Direct-Mapped Cache



6-bit memory address: 2-bit Tag, 2-bit Set Index, 2-bit Offset (4 Bytes/block).

Q: Given a memory block, which cache block is it mapped to?

A: Use 2 middle index bits in memory address to determine which cache block it is mapped to

- 00: mapped to blue block in cache
- 01: mapped to green block in cache
- 10: mapped to pink block in cache
- 11: mapped to grey block in cache

Q: Is the exact memory block in the cache?

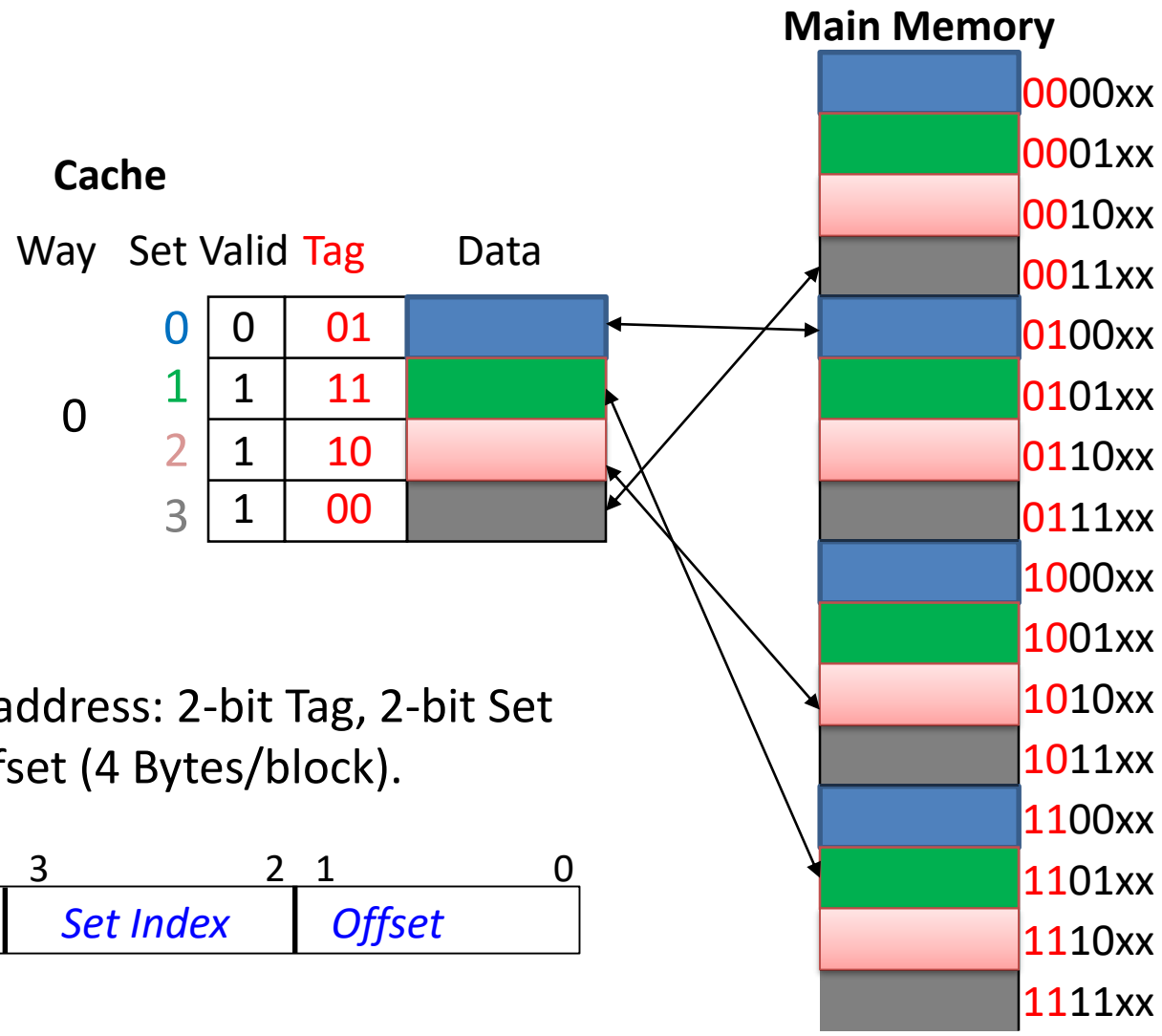
A: Compare 2 higher tag bits in memory address to the cache tag to tell if the memory block is in the cache (provided valid bit is set)

Q: Which exact Byte address in a given cache block of 4 Bytes?

A: Use the Offset

DM Cache Example

Q: Given memory address 001110, is it in the cache?

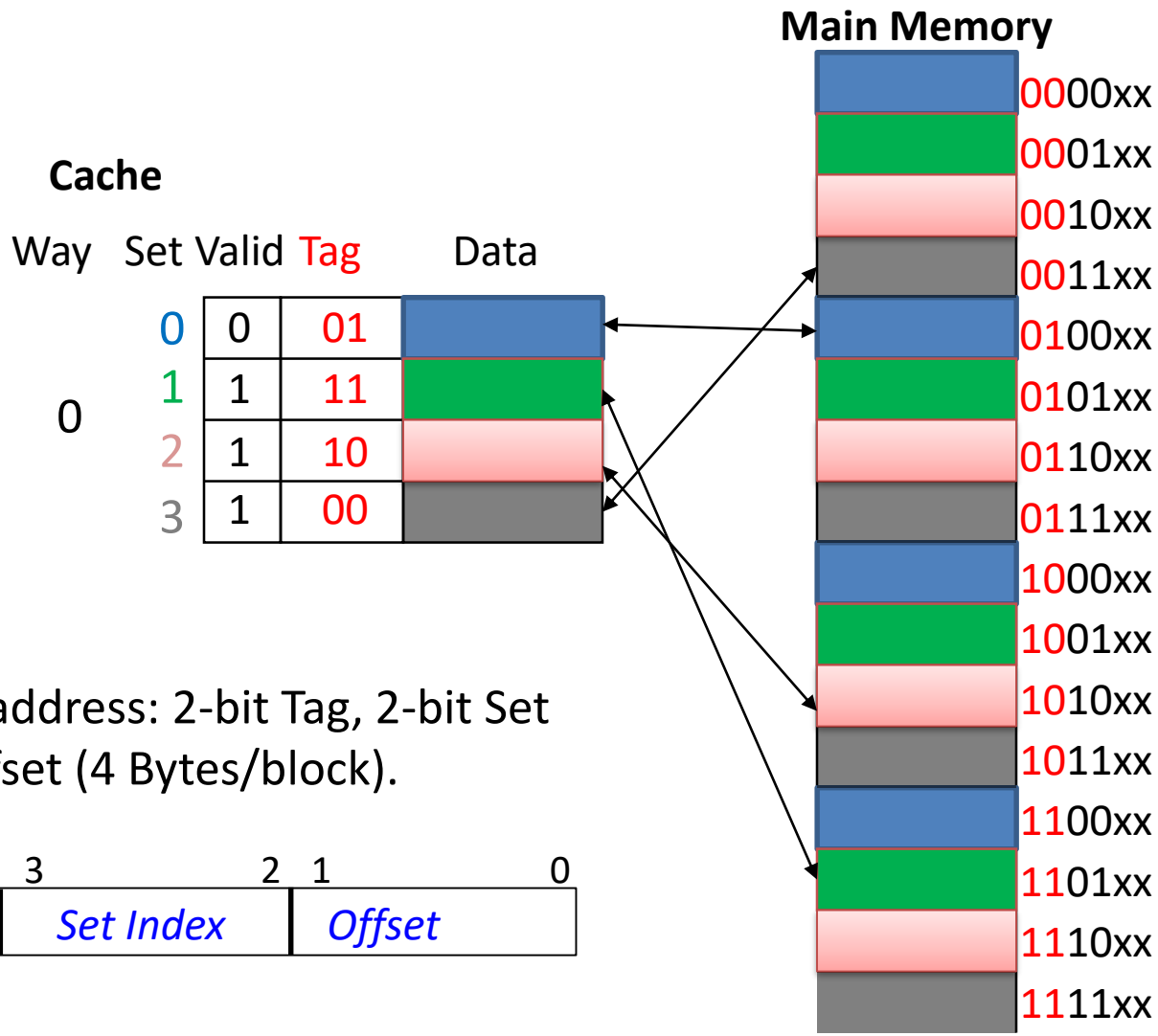


Q: Given memory address 0100xx, is it in the cache?

6-bit memory address: 2-bit Tag, 2-bit Set Index, 2-bit Offset (4 Bytes/block).

Q: Given memory address 1110xx, is it in the cache?

DM Cache Example



6-bit memory address: 2-bit Tag, 2-bit Set Index, 2-bit Offset (4 Bytes/block).

Q: Given memory address 001110, is it in the cache?

A: Yes. First, 2 middle index bits (11) means that it is mapped to grey block in cache; Second, the 2 higher tag bits (00) matches the tag in the grey block, with valid bit of 1; Finally, to get the exact Byte address, use the Offset of 10

Q: Given memory address 0100xx, is it in the cache?

A: No. First, 2 middle bits (00) means that it is mapped to blue block in cache; Second, the 2 higher tag bits (01) matches the tag in the blue block, with valid bit of 0.

Q: Given memory address 1110xx, is it in the cache?

A: No. First, 2 middle index bits (10) means that it is mapped to pink block in cache, with valid bit of 1; Second, the 2 higher tag bits (11) does not match the tag (10) in the pink block.

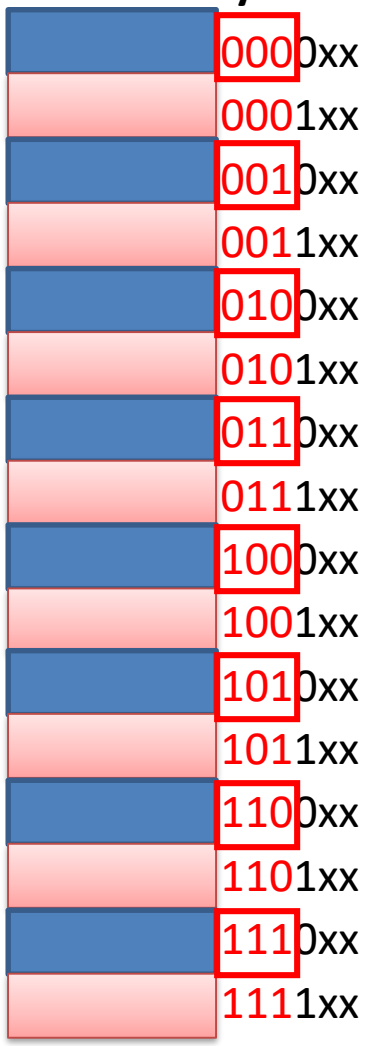
2-Way Set-Associative Cache

Each memory block is mapped to a cache set with the same color

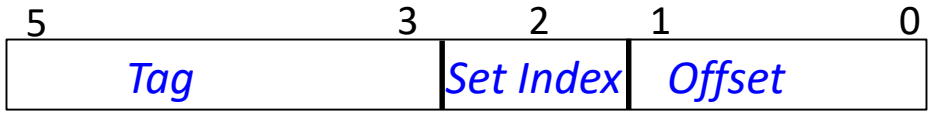
Cache

Way	Set	Valid	Tag	Data
0	0	0		
	1	1		
1	0	1		
	1	1		

Main Memory



6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Q: Given a memory block, which cache set is it mapped to?

A: Use 1 middle index bits in memory address to determine which cache set it is mapped to

- 0: mapped to blue set in cache
- 1: mapped to pink set in cache

Q: Is the exact memory block in the cache?

A: Compare 3 higher tag bits in memory address to the cache tag to tell if the memory block is in the cache (provided valid bit is set)

Q: Which exact Byte address in a given cache block of 4 Bytes?

A: Use the Offset

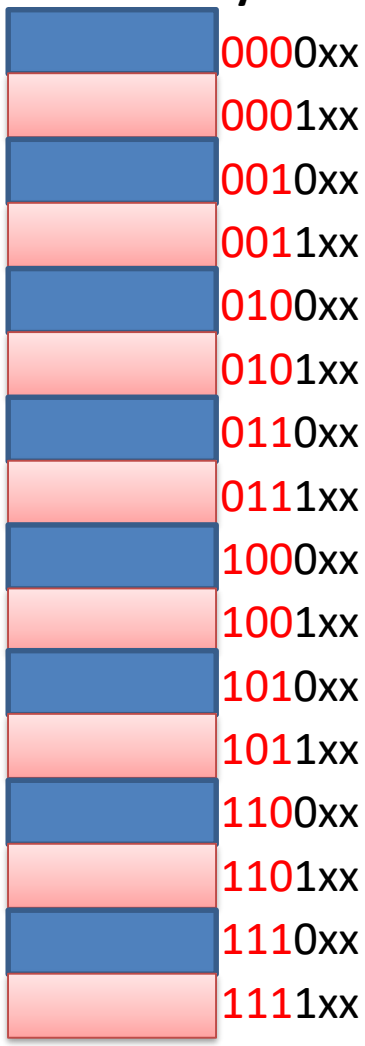
2-Way SA Cache Example

Q: Given memory address 0111xx, is it in the cache?

Cache

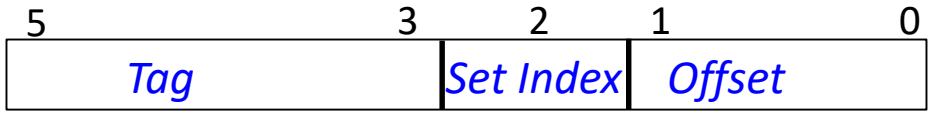
Way	Set	Valid	Tag	Data
0	0	0	010	
	1	1	111	
1	0	1	101	
	1	1	001	

Main Memory



Q: Given memory address 0100xx, is it in the cache?

6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



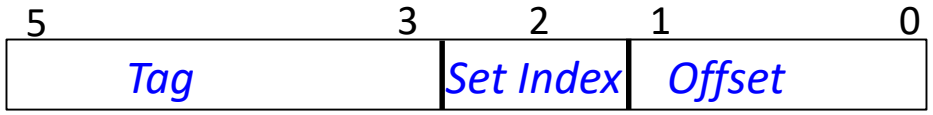
Q: Given memory address 1110xx, is it in the cache?

2-Way SA Cache Example

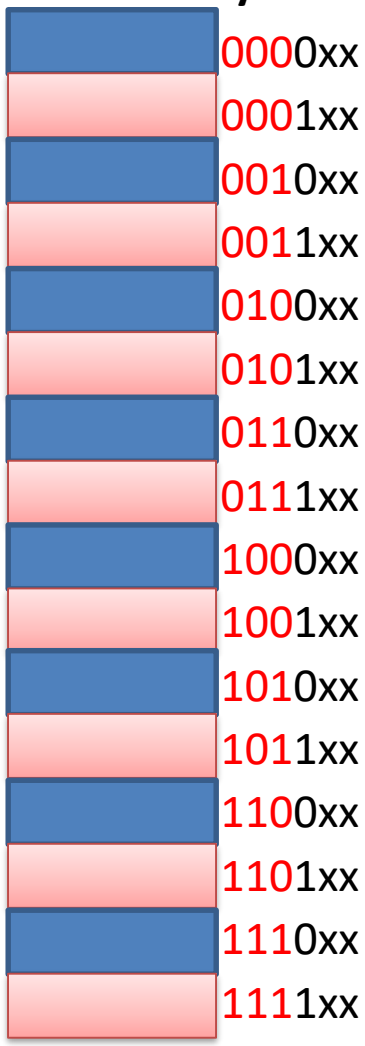
Cache

Way	Set	Valid	Tag	Data
0	0	0	010	
	1	1	101	
1	0	1	111	
	1	1	001	

6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Main Memory



Q: Given memory address 0111xx, is it in the cache?
A: No. First, 1 middle index bit (1) means that it is mapped to pink set in cache; Second, the 3 higher tag bits (011) does not match any tag in the pink set (101 and 001).
Q: Given memory address 0100xx, is it in the cache?
A: No. First, 1 middle index bit (0) means that it is mapped to blue set in cache; Second, the 3 higher tag bits (010) matches one of the tags in the blue set (010 and 111); Third, the valid bit of the corresponding cache block is 0.
Q: Given memory address 1110xx, is it in the cache?
A: Yes. First, 1 middle index bit (0) means that it is mapped to blue set in cache; Second, the 3 higher tag bits (111) matches one of the tags in the blue set (010 and 111); Third, the valid bit of the corresponding cache block is 1.

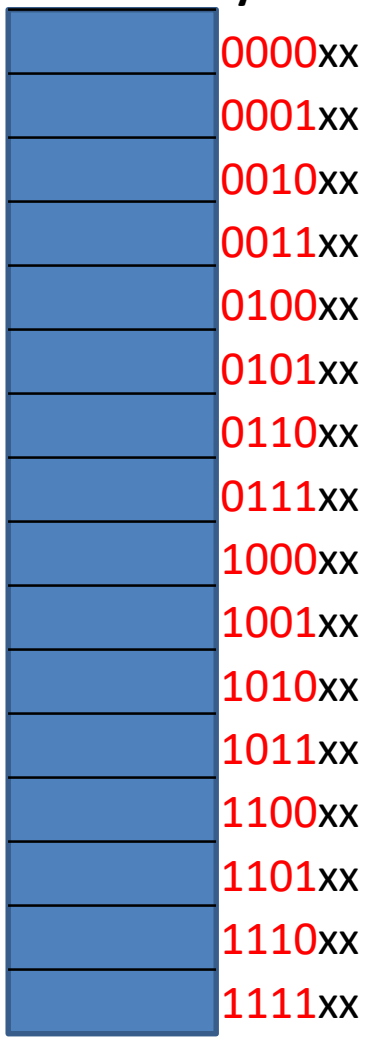
Fully-Associative Cache (4-way SA)

Each memory block is mapped to a cache set with the same color

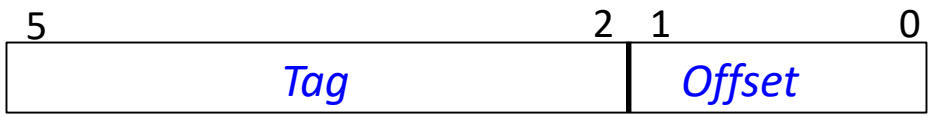
Cache

Way	Set	Valid	Tag	Data
0	0	0		
1		1		
2		1		
3		1		

Main Memory



6-bit memory address: 4-bit Tag, 0-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Q: Given a memory block, which cache set is it mapped to?
A: There is only a single blue set.

Q: Is the exact memory block in the cache?
A: Compare 4 higher tag bits in memory address to the cache tag to tell if the memory block is in the cache (provided valid bit is set)

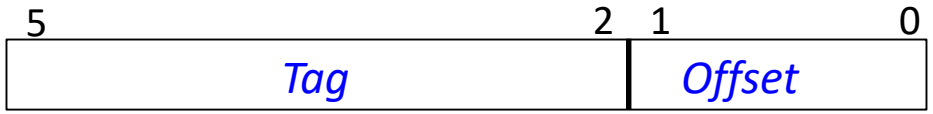
Q: Which exact Byte address in a given cache block of 4 Bytes?
A: Use the Offset

FA Cache Example

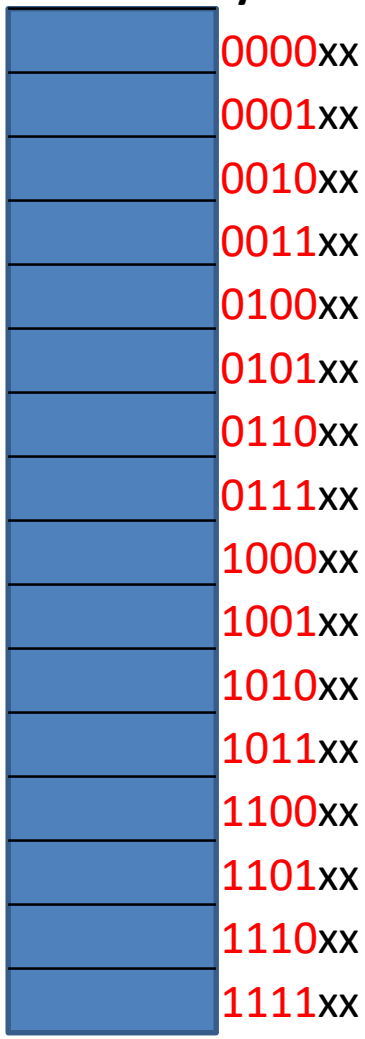
Cache

Way	Set	Valid	Tag	Data
0	0	0	0101	
1		1	1110	
2		1	1010	
3		1	0011	

6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Main Memory



Q: Given memory address 0011xx, is it in the cache?

Q: Given memory address 0100xx, is it in the cache?

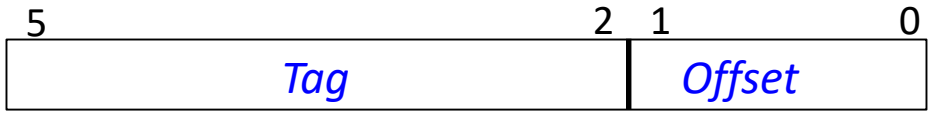
Q: Given memory address 0101xx, is it in the cache?

FA Cache Example

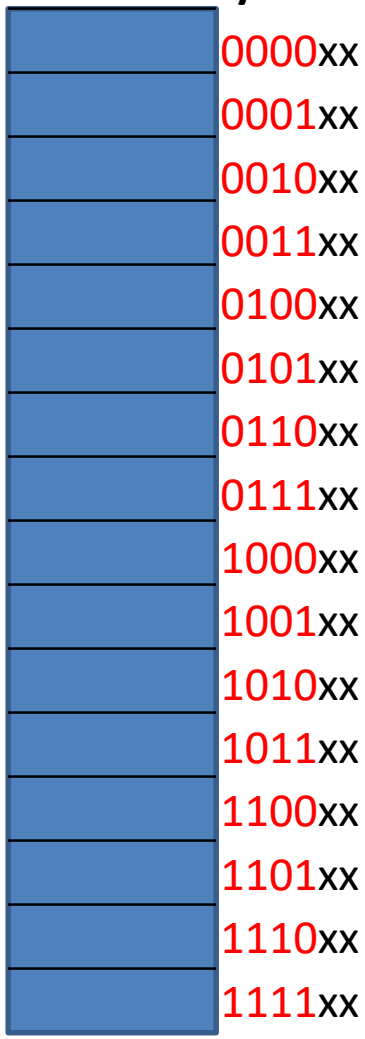
Cache

Way	Set	Valid	Tag	Data
0	0	0	0101	
1		1	1110	
2		1	1010	
3		1	0011	

6-bit memory address: 4-bit Tag, 0-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).



Main Memory



Q: Given memory address 0011xx, is it in the cache?
A: Yes. The 4 higher tag bits (0011) matches one of the tags in the blue set, and the valid bit of the corresponding cache block is 1.

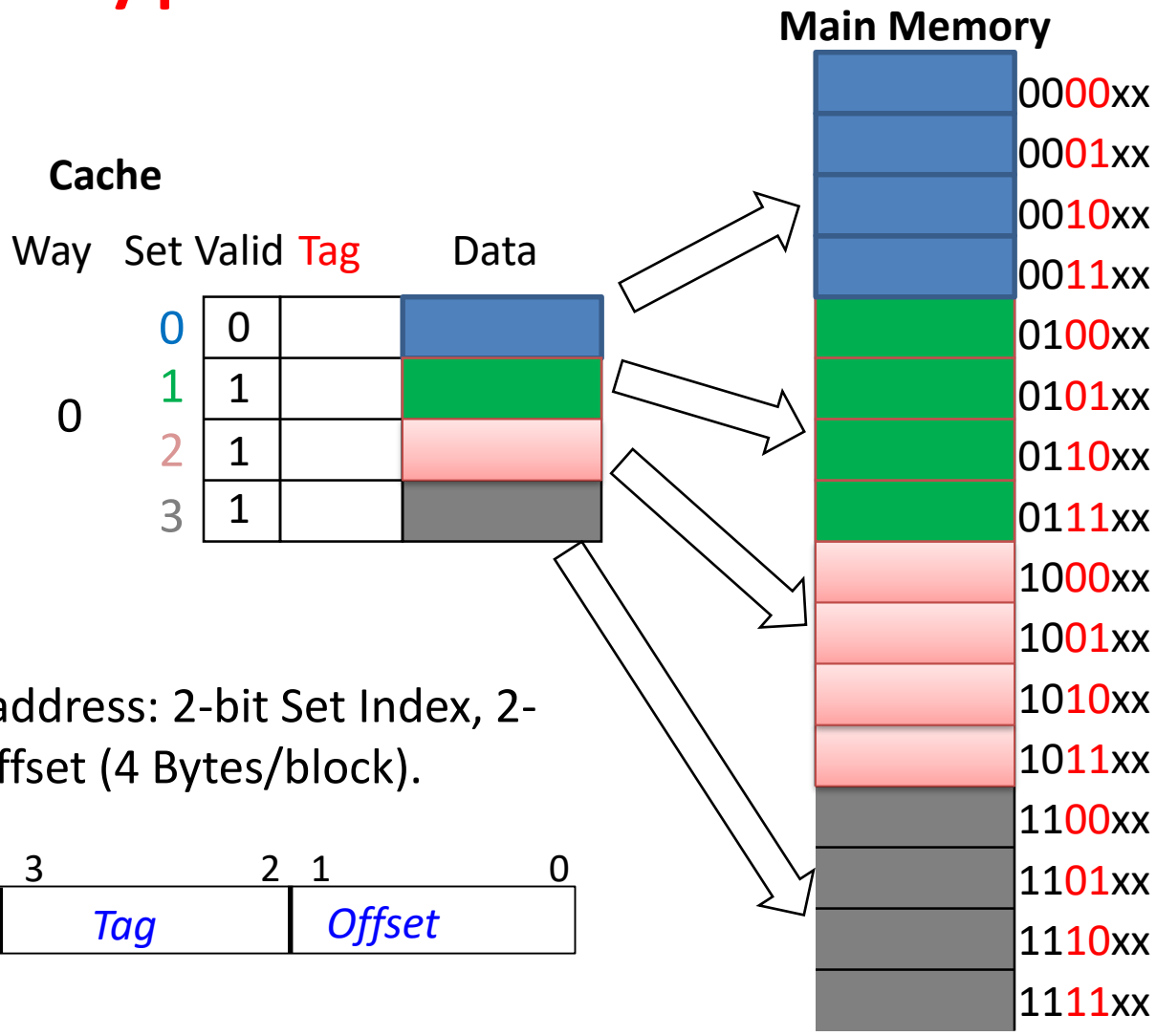
Q: Given memory address 0100xx, is it in the cache?
A: No. The 4 higher tag bits (0011) does not match any of the tags in the blue set.

Q: Given memory address 0101xx, is it in the cache?
A: No. The 4 higher tag bits (0101) matches one of the tags in the blue set, but the valid bit of the corresponding cache block is 0.

Hypothetical Cache Design

- What if we flip positions of Tag and Index? Have Index as higher-order bits, and Tag as lower-order bits?

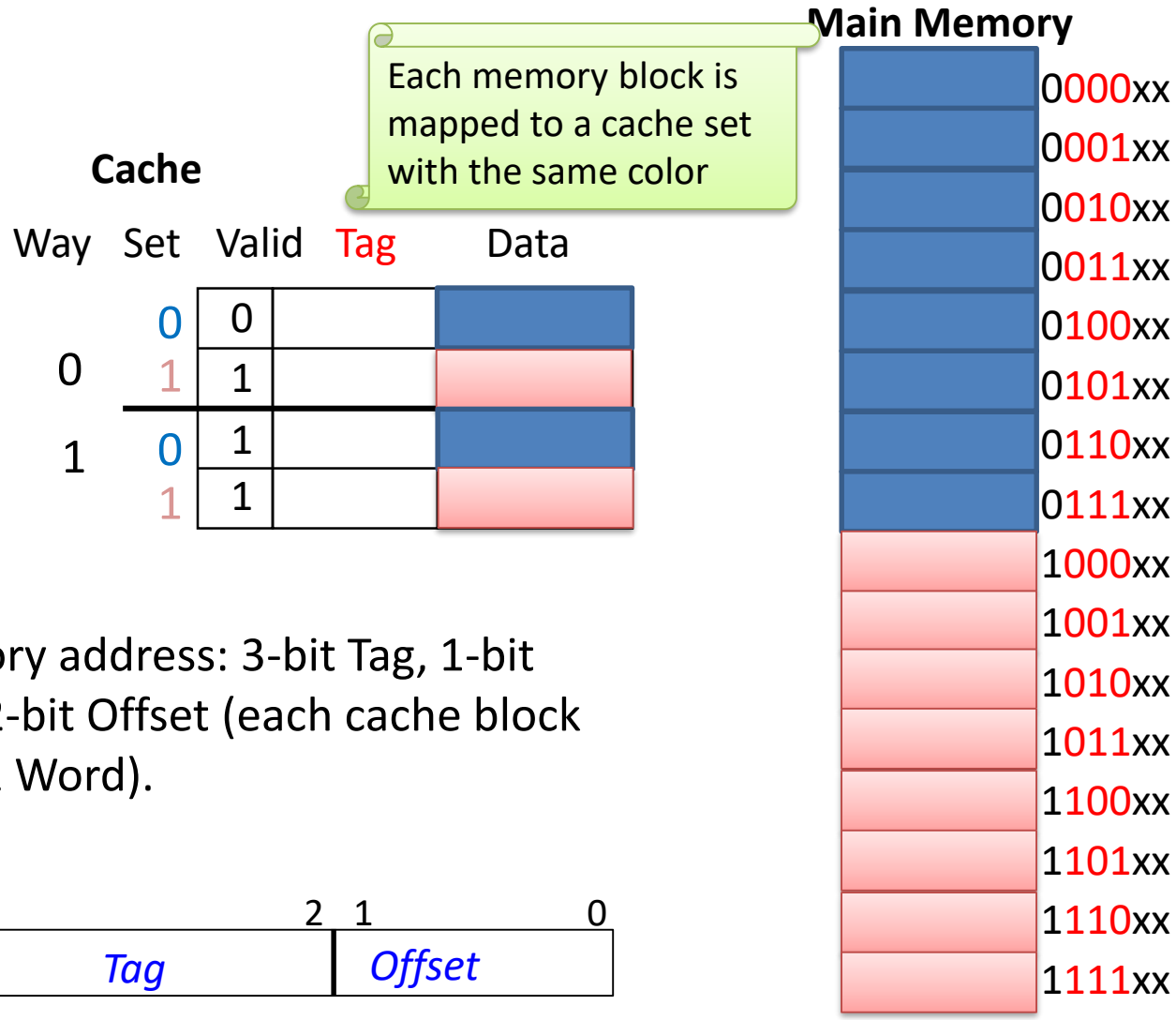
Hypothetical DM Cache Design (BAD)



6-bit memory address: 2-bit Set Index, 2-bit Tag, 2-bit Offset (4 Bytes/block).

- Bad idea: Neighboring blocks in memory are mapped to the same cache line, since they share the same Set 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.

Hypothetical 2-Way SA Cache Design



6-bit memory address: 3-bit Tag, 1-bit Set Index, 2-bit Offset (each cache block is 4 Bytes/1 Word).

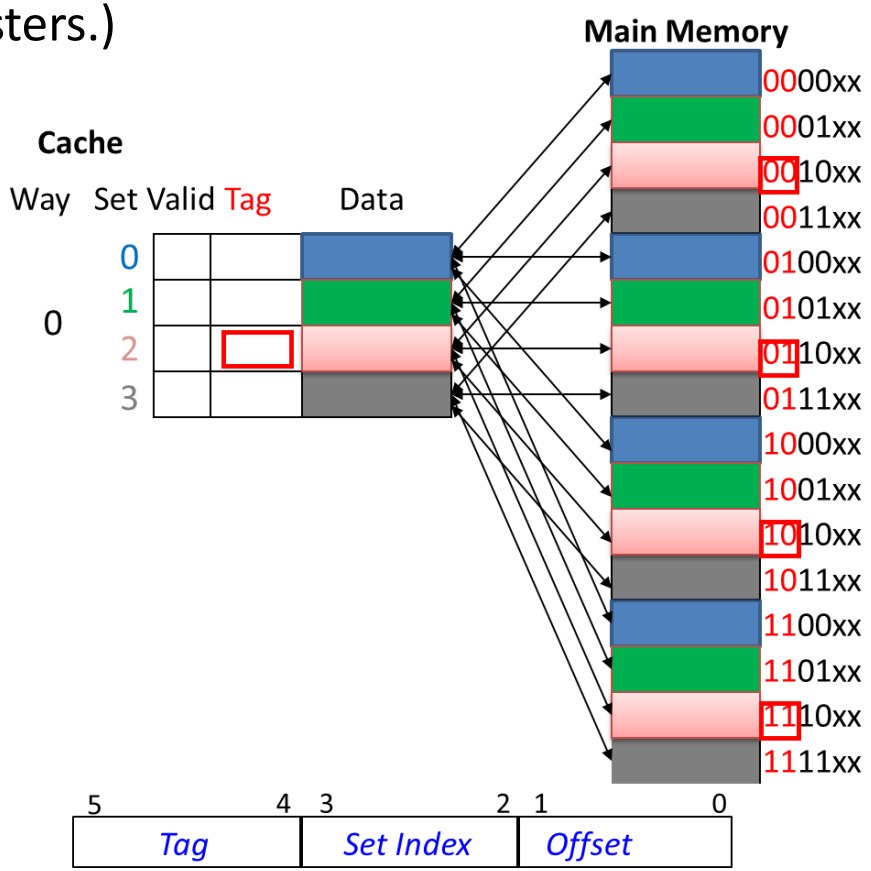
- Still a likely bad idea, but not as bad as the DM cache in the last slide
  - Each set (color) has 2 possible addresses in the cache
- As associativity increases, this Tag-in-the-Middle approach becomes more feasible

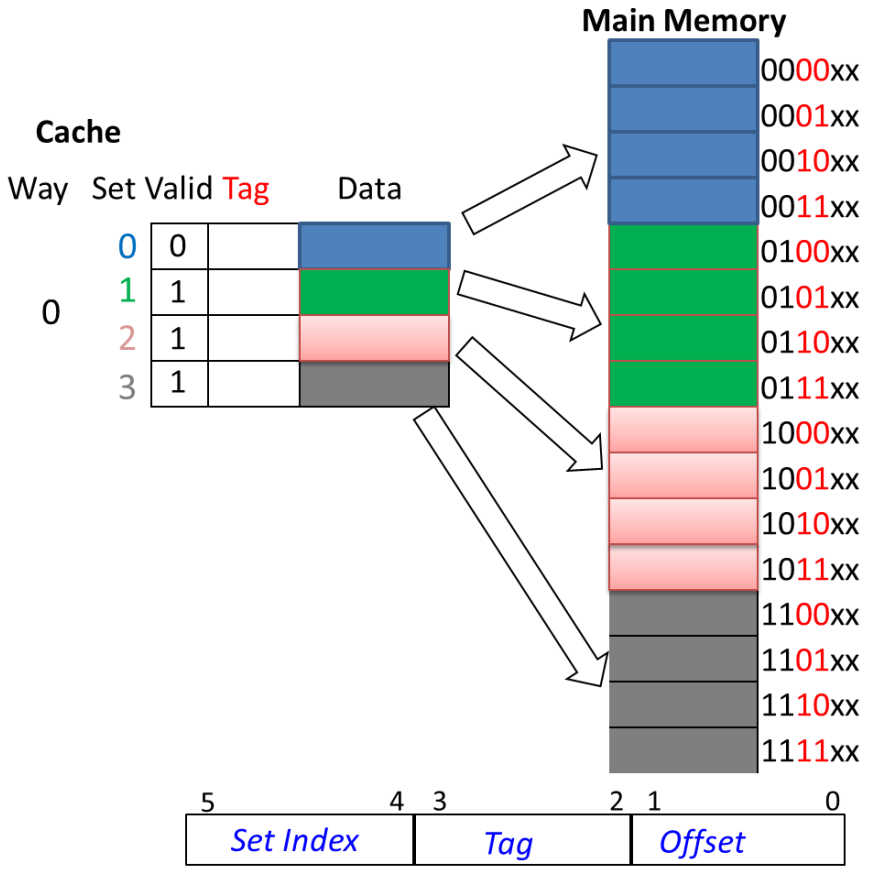
Quiz: # Cache Misses for DM Cache

- Consider the following (silly) program that repeatedly accesses an array of 4-Byte ints A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a DM cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model? (Assuming that LDR is used to load elements of A[] in every iteration, instead of keeping them in CPU registers.)





Answer: # Cache Misses for DM Cache

- Consider the following program that repeatedly accesses an array of words A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a DM cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model?

For Tag-Set Index-Offset model: 4 misses

- 1st cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2nd cache miss brings into cache a block with address 0001xx into Set 1, which contains A[1]
- 3rd cache miss brings into cache a block with address 0010xx into Set 2, which contains A[2]
- 4th cache miss brings into cache a block with address 0011xx into Set 3, which contains A[3]
- 9996 cache hits.

Tag-Set Index-Offset model has better spatial locality!

For Set Index-Tag-Offset model: 10000 misses

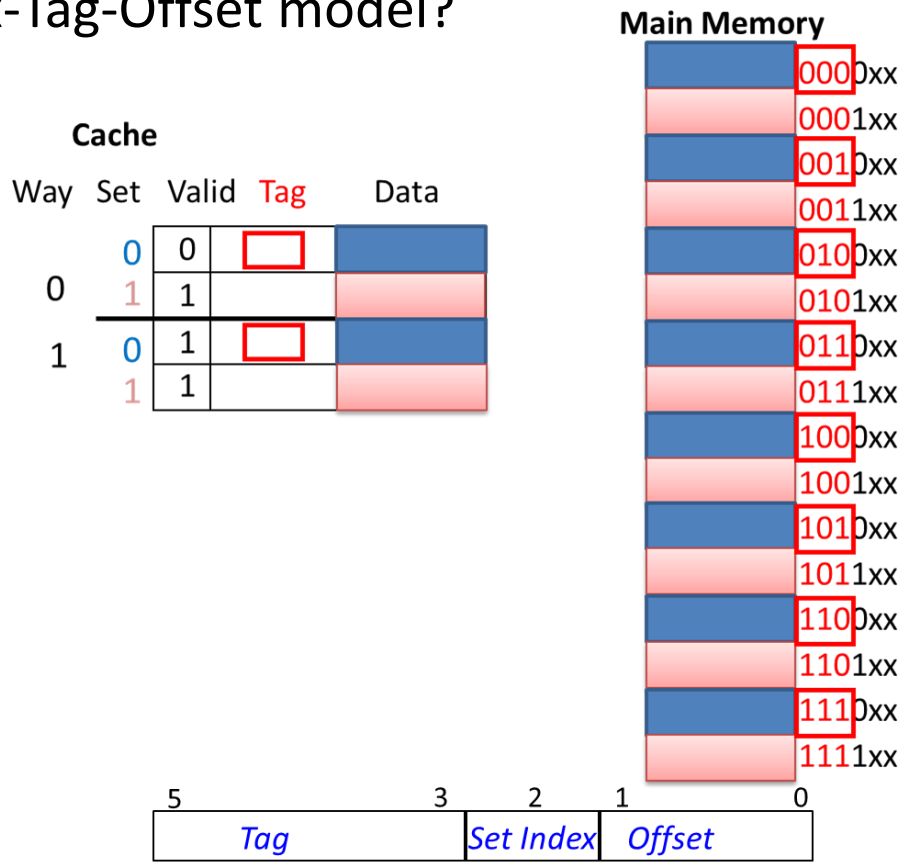
- 1st cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2nd cache miss brings into cache a block with address 0001xx into Set 0, which contains A[1], and replaces A[0]
- 3rd cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2], and replaces A[1]
- 4th cache miss brings into cache a block with address 0011xx into Set 0, which contains A[3], and replaces A[2]
- 5th cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0], and replaces A[3]
-

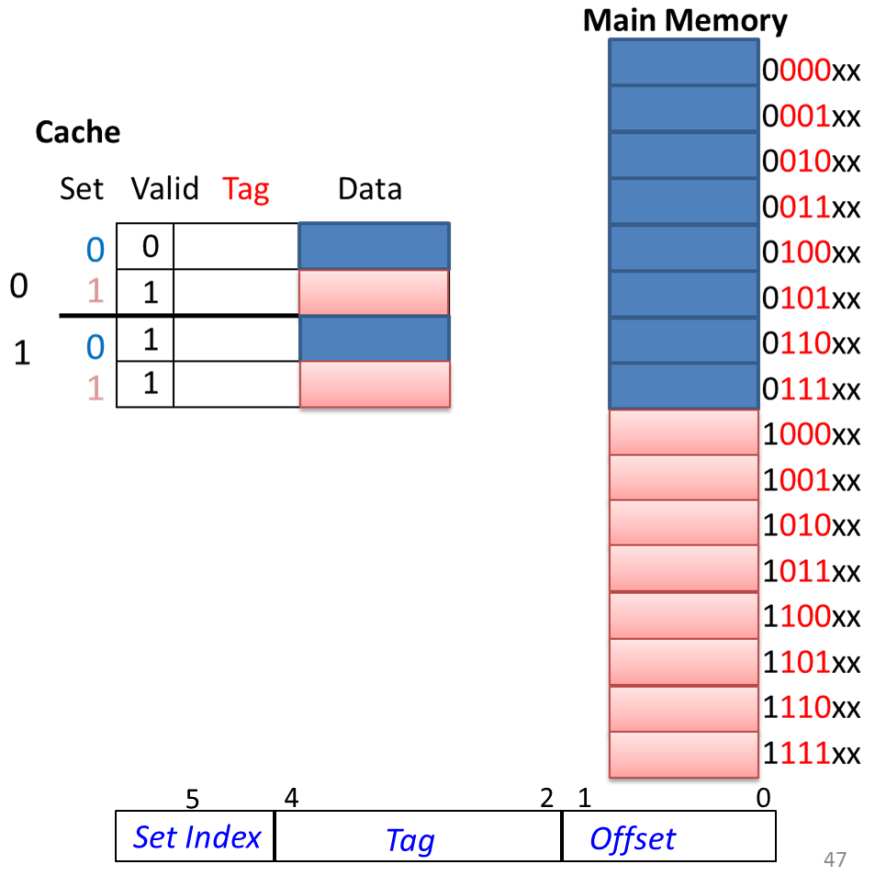
Quiz: # Cache Misses for 2-Way SA Cache

- Consider the following program that repeatedly accesses an array of words A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a 2-way SA cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model?





Answer: # Cache Misses for 2-Way SA Cache

- Consider the following program that repeatedly accesses an array of words A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a 2-way SA cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model?

For Tag-Set Index-Offset model: 4 misses

- 1st cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2nd cache miss brings into cache a block with address 0001xx into Set 1, which contains A[1]
- 3rd cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2]
- 4th cache miss brings into cache a block with address 0011xx into Set 1, which contains A[3]
- 9996 cache hits.

Tag-Set Index-Offset model has better spatial locality!

For Set Index-Tag-Offset model: 10000 misses for LRU (Least-Recently-Used) replacement policy

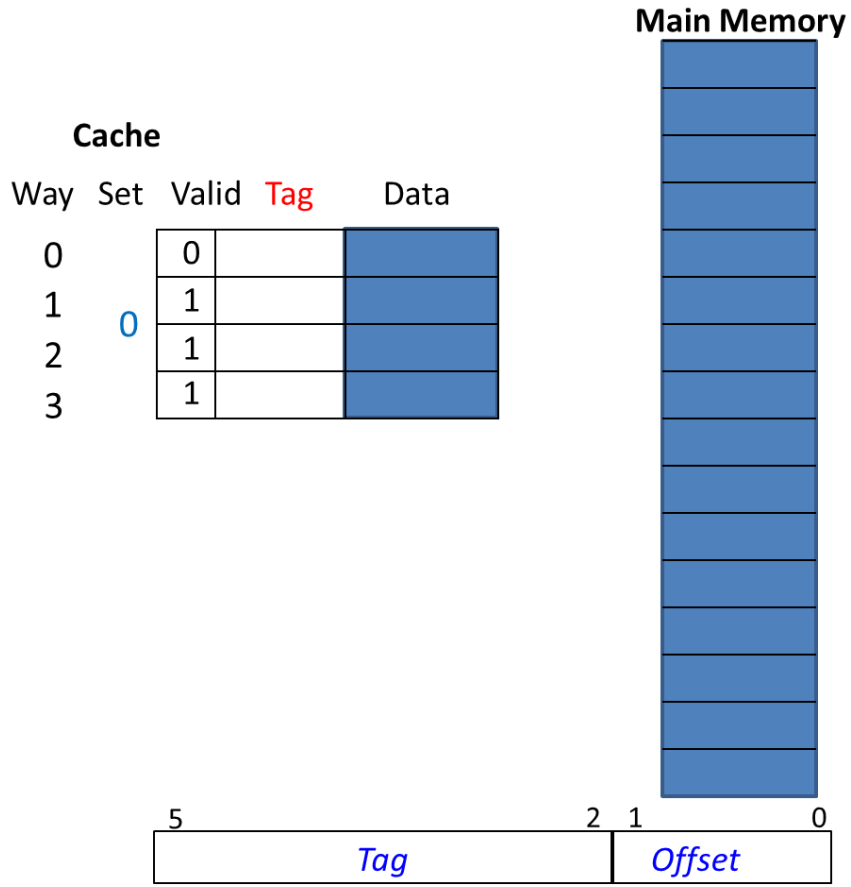
- 1st cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2nd cache miss brings into cache a block with address 0001xx into Set 0, which contains A[1]
- 3rd cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2], and replaces A[0].
- 4th cache miss brings into cache a block with address 0011xx into Set 0, which contains A[3], and replaces A[1].
- 5th cache miss brings into cache a block with address 0000xx into Set 0, and replaces A[2].
- Similar process repeats for Set 1.
- (MRU (Most-Recently-Used) replacement policy can keep A[0] and A[2] in cache, leading to lower Miss rate. Details omitted)

Quiz: # Cache Misses for FA Cache

- Consider the following program that repeatedly accesses an array of words A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a FA cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model?



Answer: # Cache Misses for FA Cache

- Consider the following program that repeatedly accesses an array of words A[4]

  for (int i=0; i++, i<10000) {sum += A[0]+A[1]+A[2]+A[3];}

- Suppose A[0] starts at memory address 000000, how many cache misses due to reading array A[] for a FA cache with either 1) the Tag-Set Index-Offset model; or 2) the Set Index-Tag-Offset model?

For either model: 4 misses

- 1st cache miss brings into cache a block with address 0000xx into Set 0, which contains A[0]
- 2nd cache miss brings into cache a block with address 0001xx into Set 0, which contains A[1]
- 3rd cache miss brings into cache a block with address 0010xx into Set 0, which contains A[2]
- 4th cache miss brings into cache a block with address 0011xx into Set 0, which contains A[3]
- 9996 cache hits.

Trick question! There is no Set Index, so the two models are the same!

Key Equations

sets = $2^{\text{SI size}}$; # Bytes/block=$2^{\text{Offset size}}$

blocks = # ways (associativity) * # sets

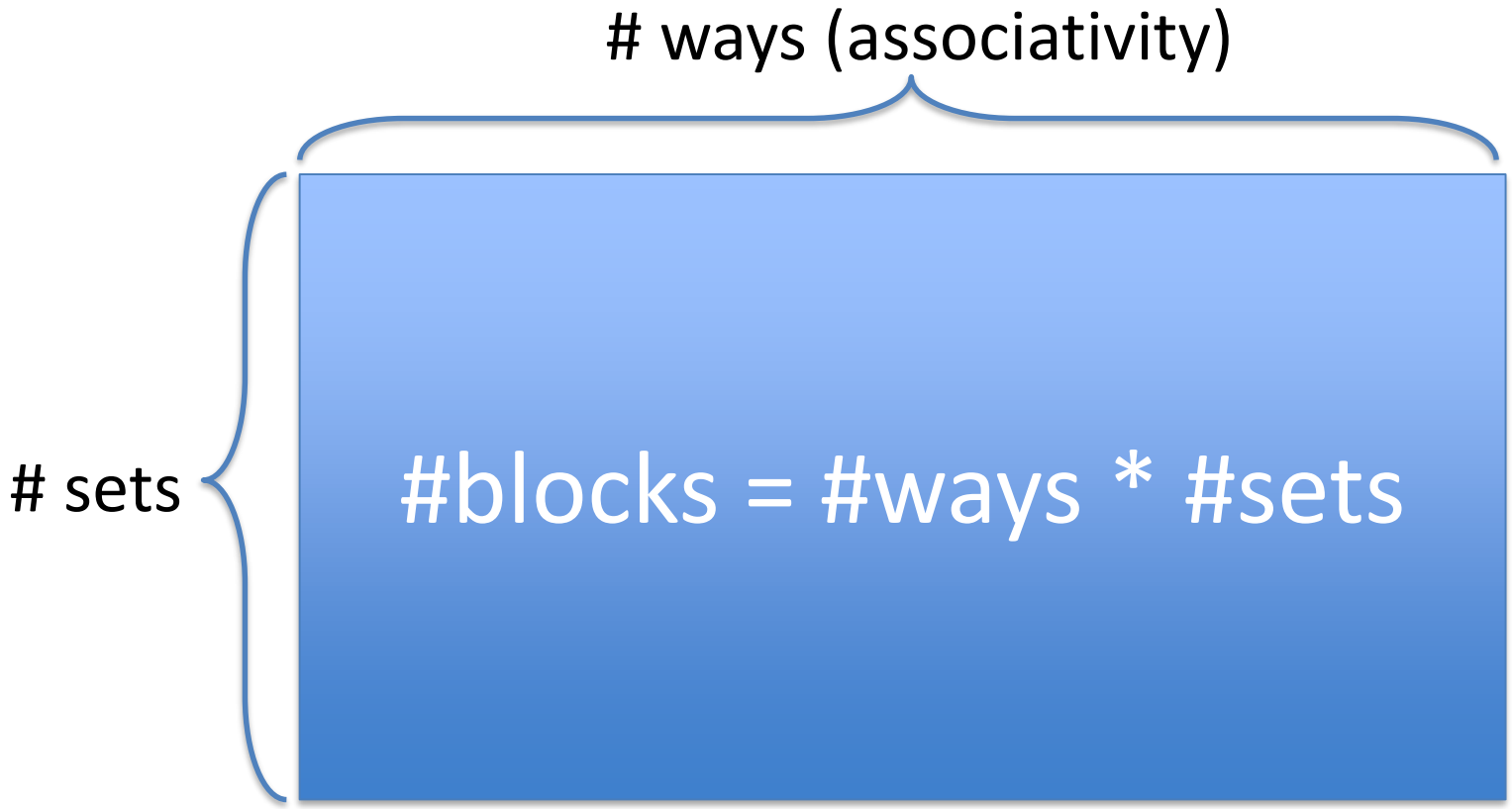
cache capacity = # blocks * # Bytes/block

Tag size does not affect cache capacity; depends on memory address length

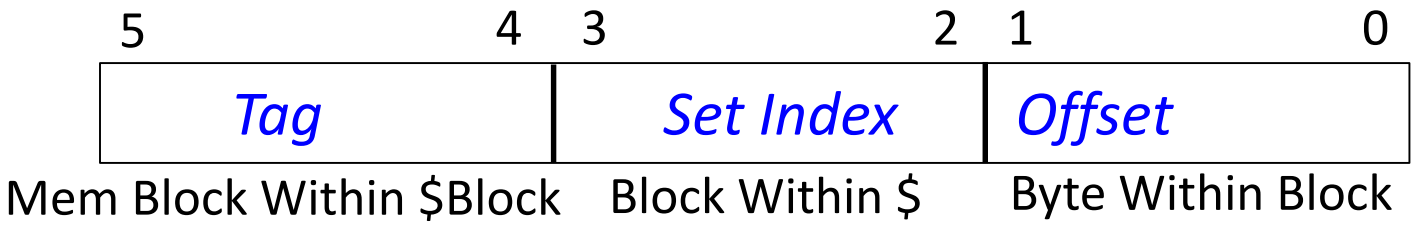
SI size determines # sets = $2^{\text{SI size}}$

Offset size determines Bytes/block = $2^{\text{Offset size}}$

2D Visualization of Cache Organization



Cache Example

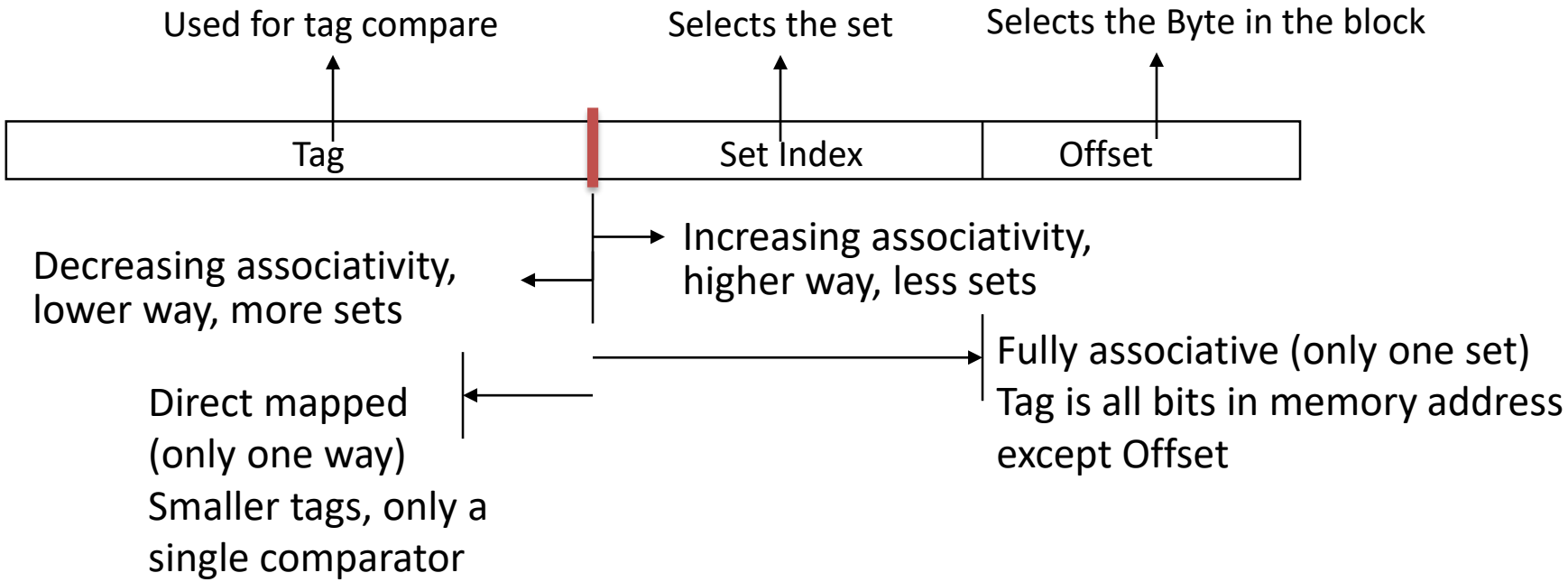


- Assume: DM cache; 6-bit memory address: 2-bit Tag, 2-bit index, 2-bit Offset. Compute cache capacity and memory size.
  - 2-bit Offset => Bytes/block = 4;
  - # sets = $2^{\text{SI Size}}$ = 4
  - # cache blocks = # ways * # sets = 1*4 = 4
  - cache capacity = # cache blocks * Bytes/block = 4*4 = 16B
- Memory size: 2^4 (2-bit tag +2-bit SI) = 16 blocks = 64 Bytes

Recall:

sets = $2^{\text{SI size}}$; # Bytes/block=$2^{\text{Offset size}}$
blocks = # ways (associativity) * # sets
cache capacity = # blocks * # Bytes/block

Range of SA Caches



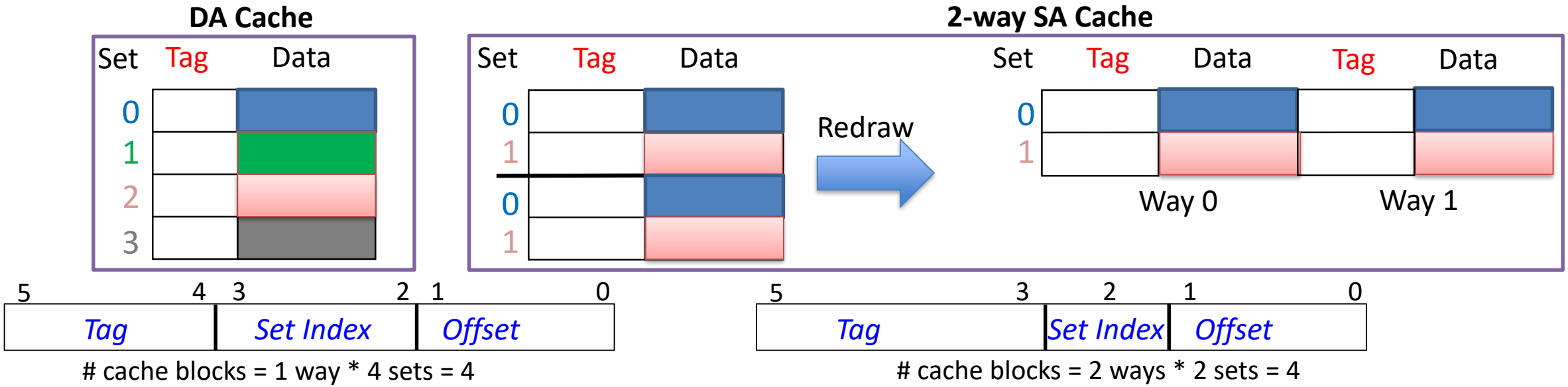
- memory_address_size = tag_size (T) + set_index_size (SI) + block_offset_size (BO)
- For a fixed-size cache, and a given block size, if we decrease the size of the index by 1 bit and increases the size of the tag by 1 bit (pushing the red bar to the right by 1 bit) :
  - Doubled: #ways = #blocks per cache set = associativity
  - Halved: #cache sets
- Quiz: what if Tag has 0 bits?
  - Then cache has same size as memory → clearly unrealistic!

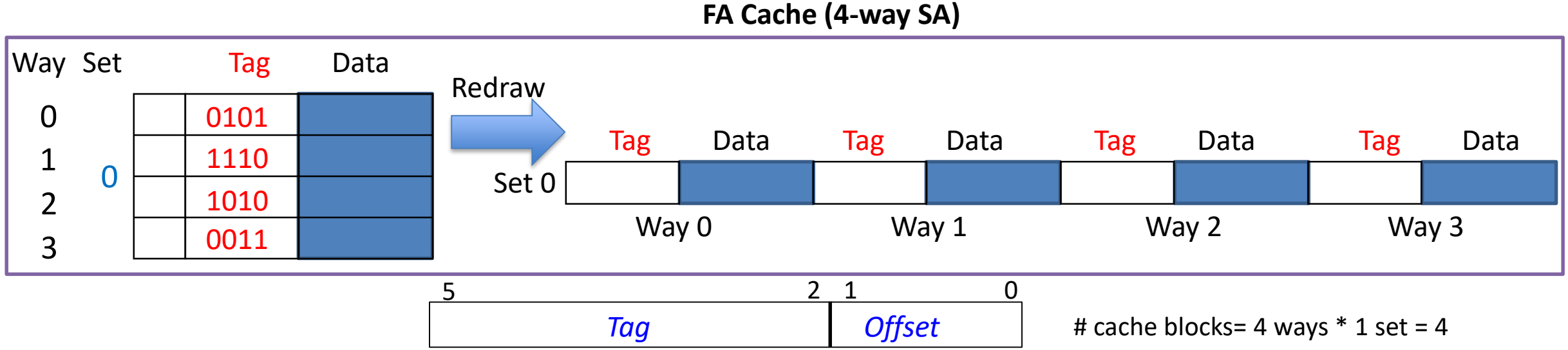
Review

Alternative Cache Organizations

- A memory block is mapped to one cache set, which may contain one or more cache blocks
- Direct Mapped (DM)
  - Each cache set has 1 cache block; # cache sets = # cache blocks
  - A memory block is mapped to 1 possible cache block
- Fully Associative (FA)
  - A single cache set contains all cache blocks; # cache sets = 1
  - A memory block can be mapped to any cache block
- N-way Set Associative (SA)
  - Each cache set has N cache blocks; # cache sets = # cache blocks / N
  - N is also called associativity
  - A memory block can be mapped to one of N possible cache blocks
- DM and FA are special cases of SA
  - DM = 1-way SA (N = 1)
  - FA = N-way SA (N = total number of cache blocks)

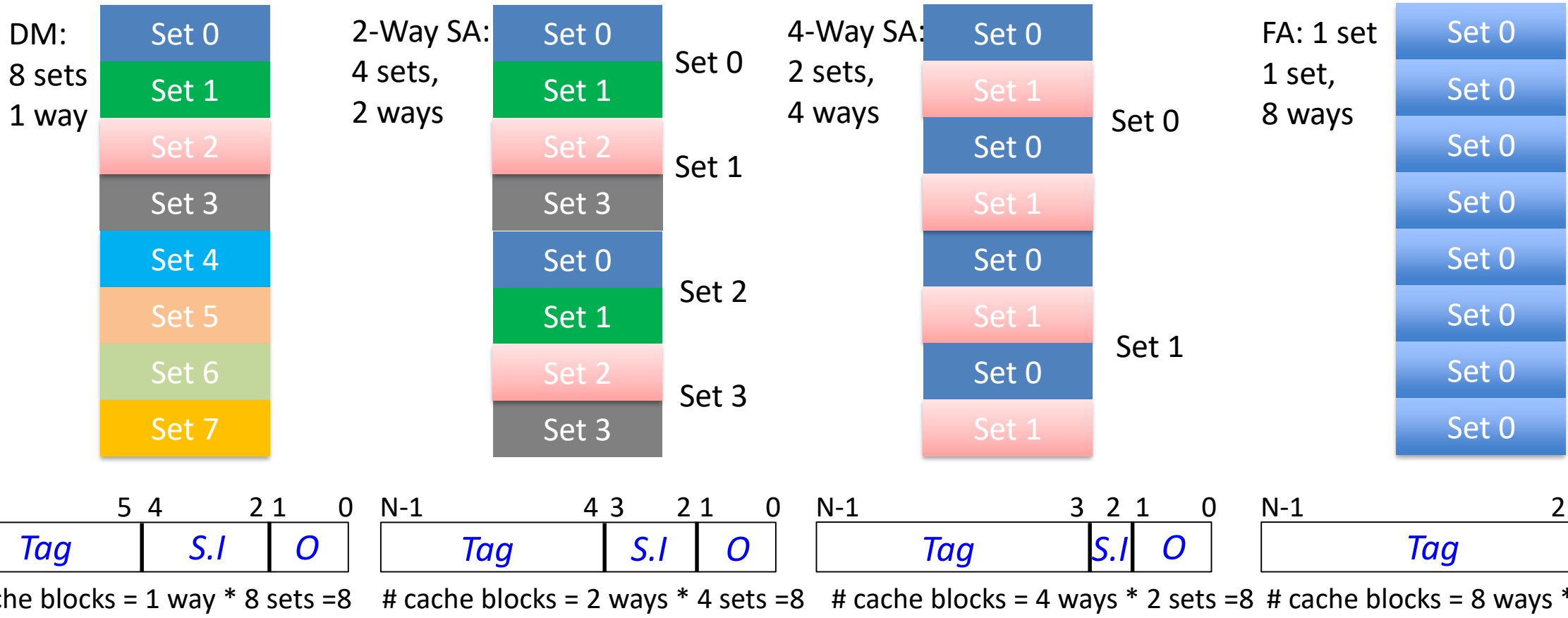
4-Block Cache Summary (Valid Bit Omitted)



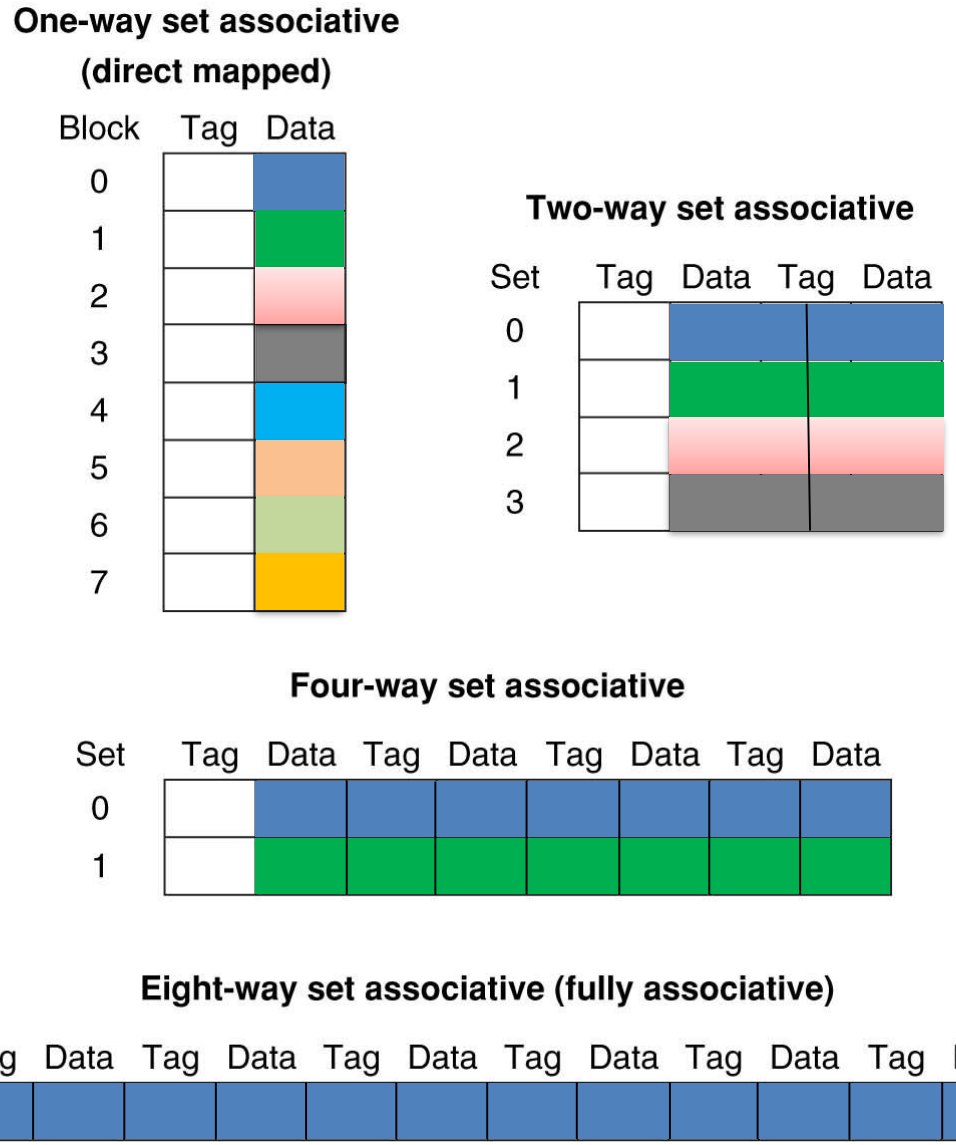


8-Block Cache

- Each color denotes a cache set



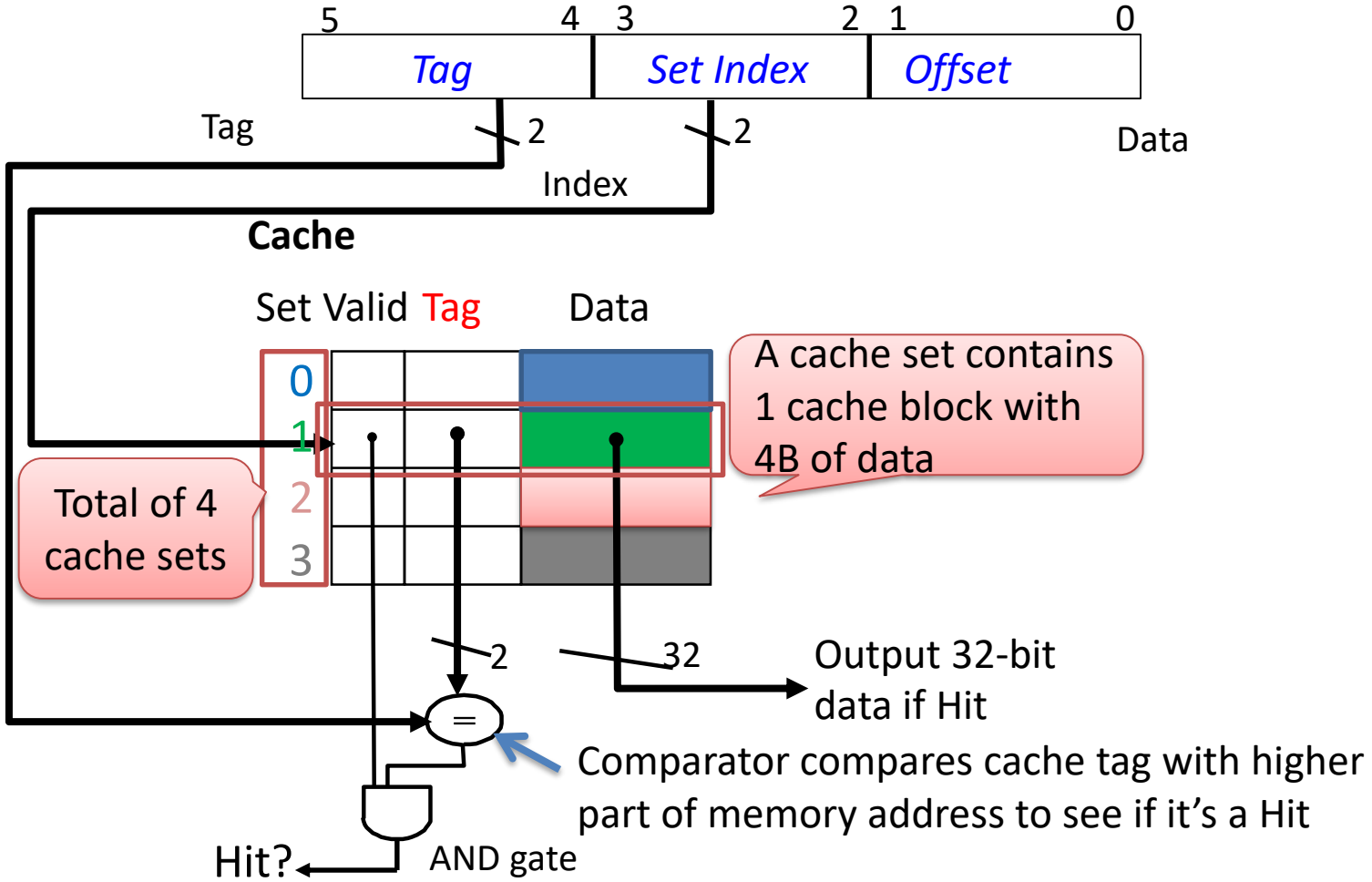
8-Block Cache Summary



- More ways (associativity) → fewer cache sets → cache structure is more "short (vertically) and fat (horizontally)"
- Fewer ways (associativity) → more cache sets → cache structure is more "tall (vertically) and skinny (horizontally)"

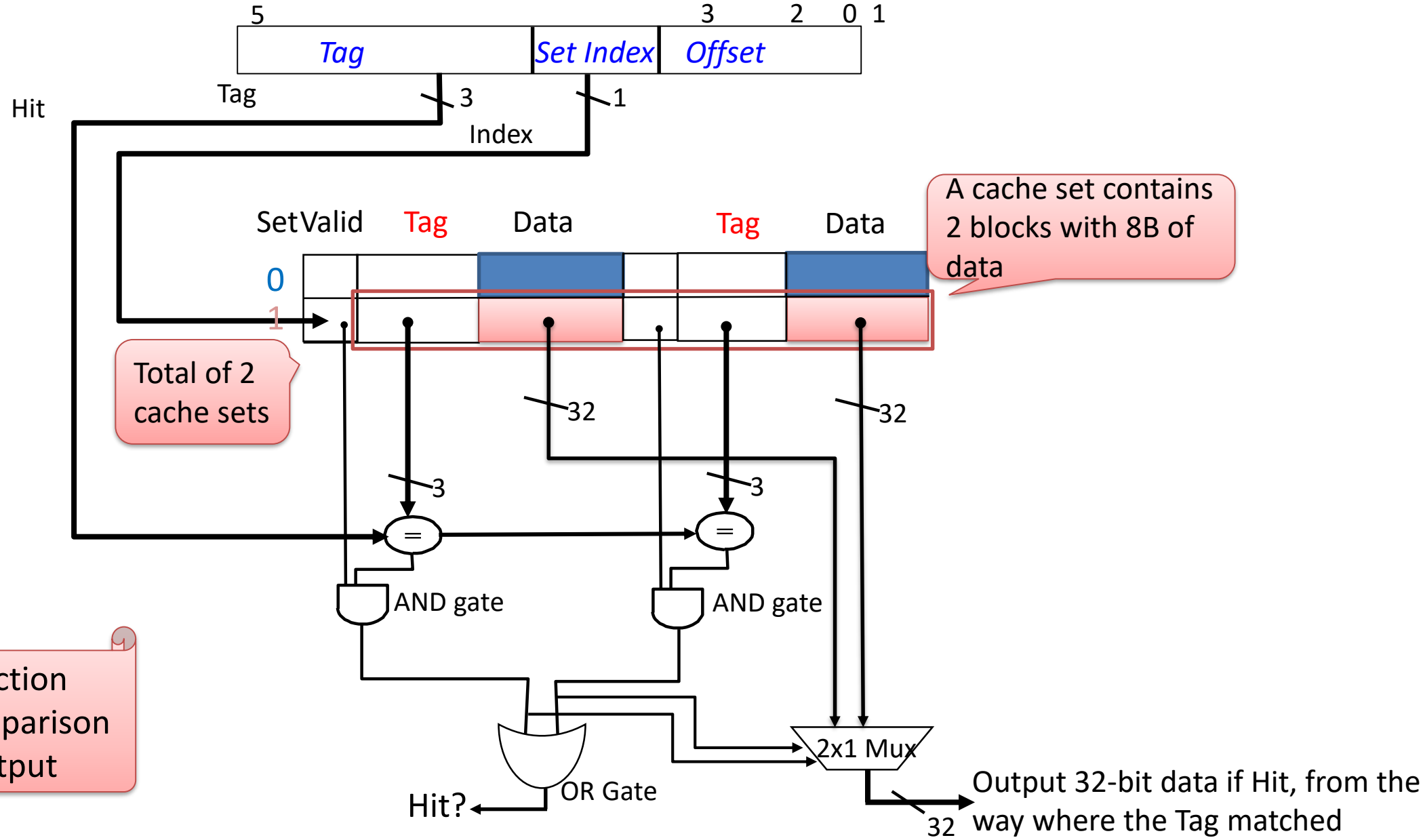
DM Cache Hardware (6-bit Memory)

- 1. Set selection: the cache set index is used to select a set (1 block) from the cache.
- 2. Tag comparison: The tag of that block is compared with the higher bits of the memory address with HW comparator
- 3. Data output: If the tag matches, and the block's Valid bit is 1, we have a cache hit; select and output 32-bit data of that cache block; Otherwise, we have a cache miss.



1. Set selection
2. Tag comparison
3. Data output

2-way SA Cache Hardware (6-bit Memory)



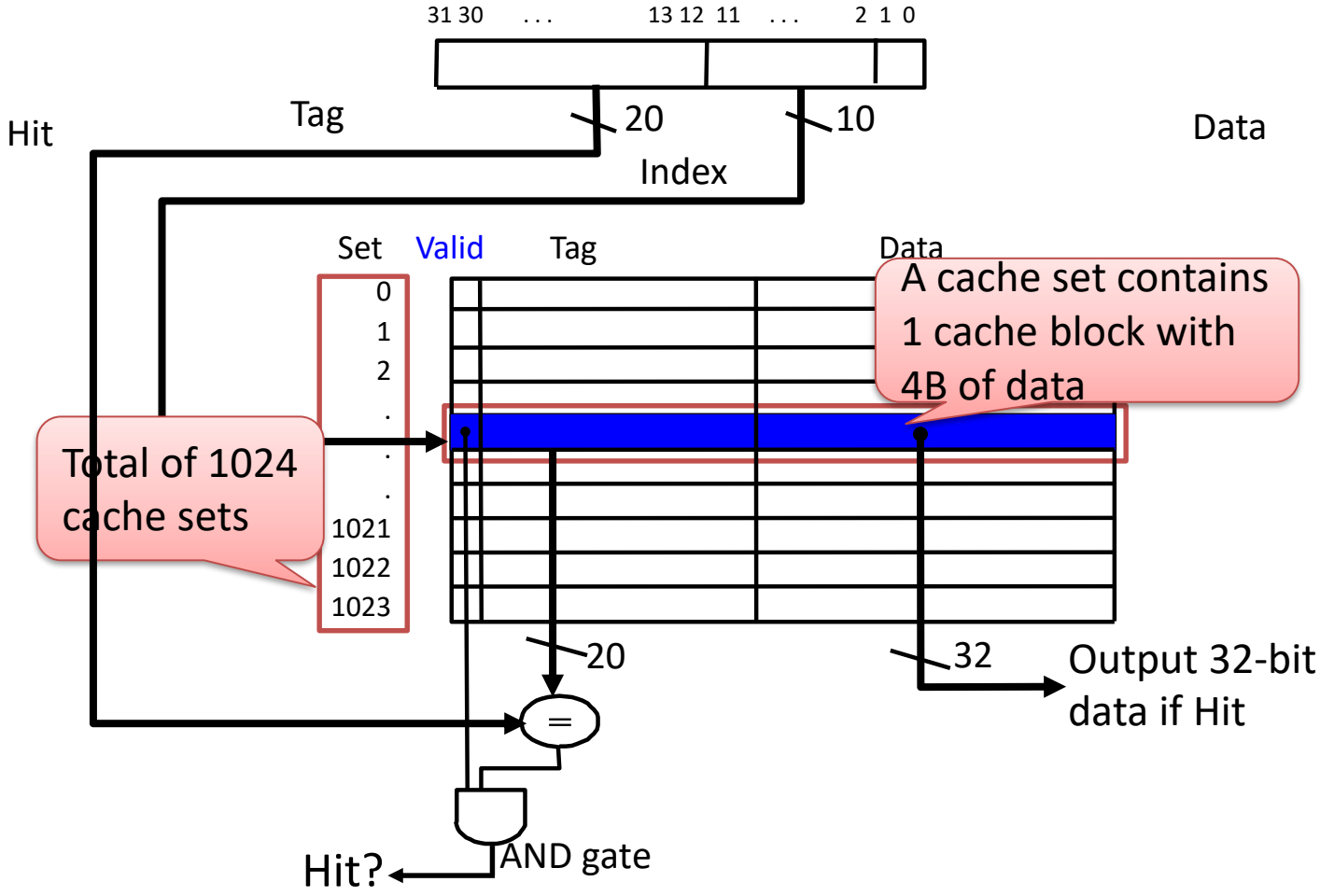
1. Set selection
2. Tag comparison
3. Data output

Notes on 2-Way SA Cache

- 1. Set selection: the cache set index is used to select a set (2 blocks) from the cache.
- 2. Tag comparison: Tags of the 2 blocks in the set are compared with the higher bits of the memory address with 2 HW comparators in parallel
- 3. Data output: If one of the tags matches (the OR gate outputs true if any input is true), and that block's Valid bit is 1, we have a cache hit; select and output 32-bit data of that cache block from the way where the tag matched, with 2x1 Mux. Otherwise, we have a cache miss
  - The cache controller hardware always returns a 32-bit word; the application software can use the Offset bits for Byte-addressing into the word.
- Think of 2-way SA cache as two DM caches working in parallel
  - More hardware needed than DM cache: 2 comparators; 1 Mux

DM Cache Hardware (32-bit Memory)

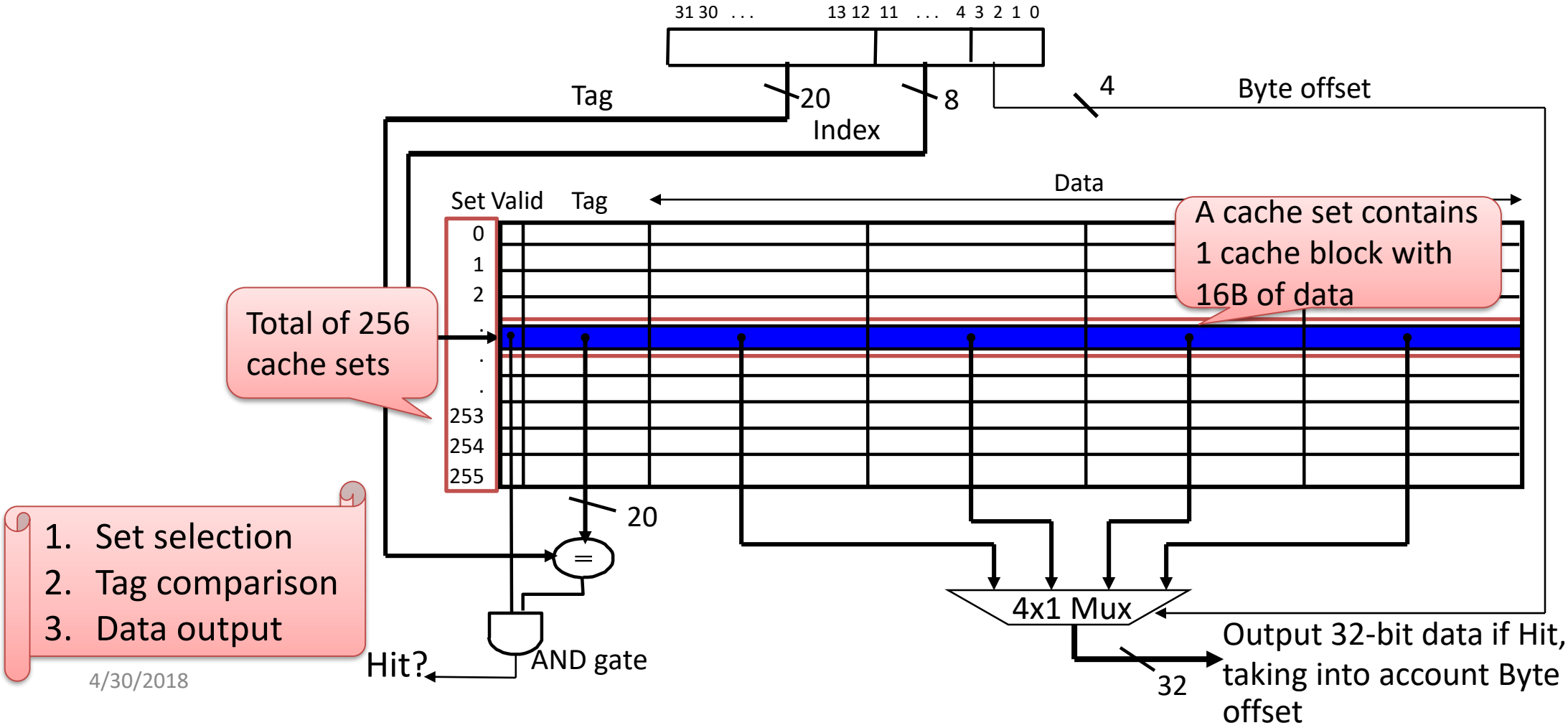
- 32-bit memory address: 20-bit Tag, 10-bit set index, 2-bit Offset (each cache block is 4 Bytes).
  - Cache size: 1 way * 2^{10} (10-bit set index) = 1K blocks = 1K Words (4 KB)
  - Memory size: 2^{30} (20-bit tag +10-bit set index) =1G blocks = 1G Words (4 GB)



1. Set selection
2. Tag comparison
3. Data output

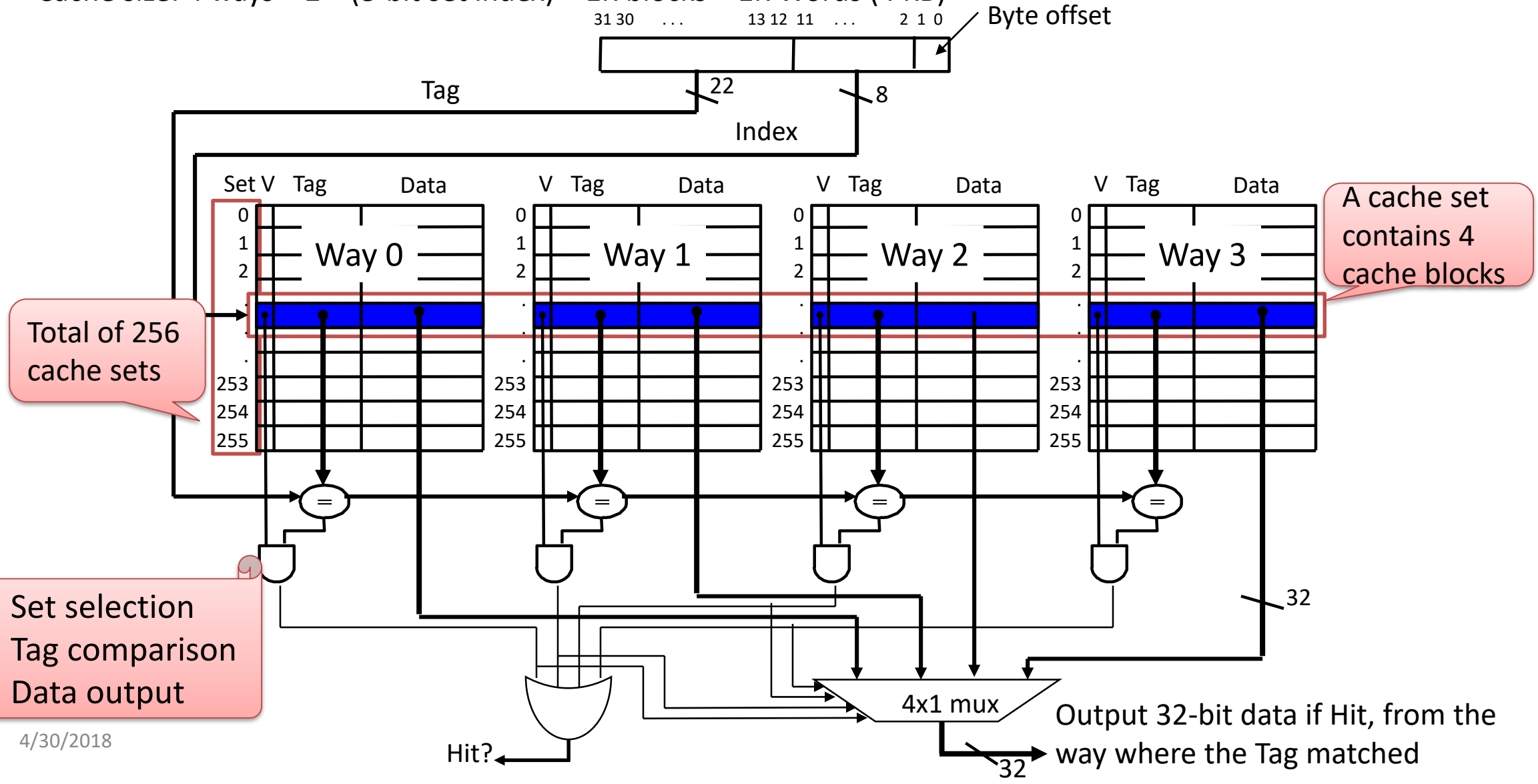
Multi-Word Block DM Cache

- 32-bit memory address: 20-bit Tag, 8-bit set index, 4-bit Offset (each cache block is 16 Bytes/4 Words).
  - Cache size: 1 way * 2^{8} (8-bit set index) = .25K blocks = 1K Words (4 KB)
  - Memory size: 2^{28} (20-bit tag +8-bit set index) =.25G blocks = 1G Words (4 GB)
- A 4x1 Mux is needed to select a 32-bit word from 4 words in the hit cache block, controlled by the 4-bit Offset



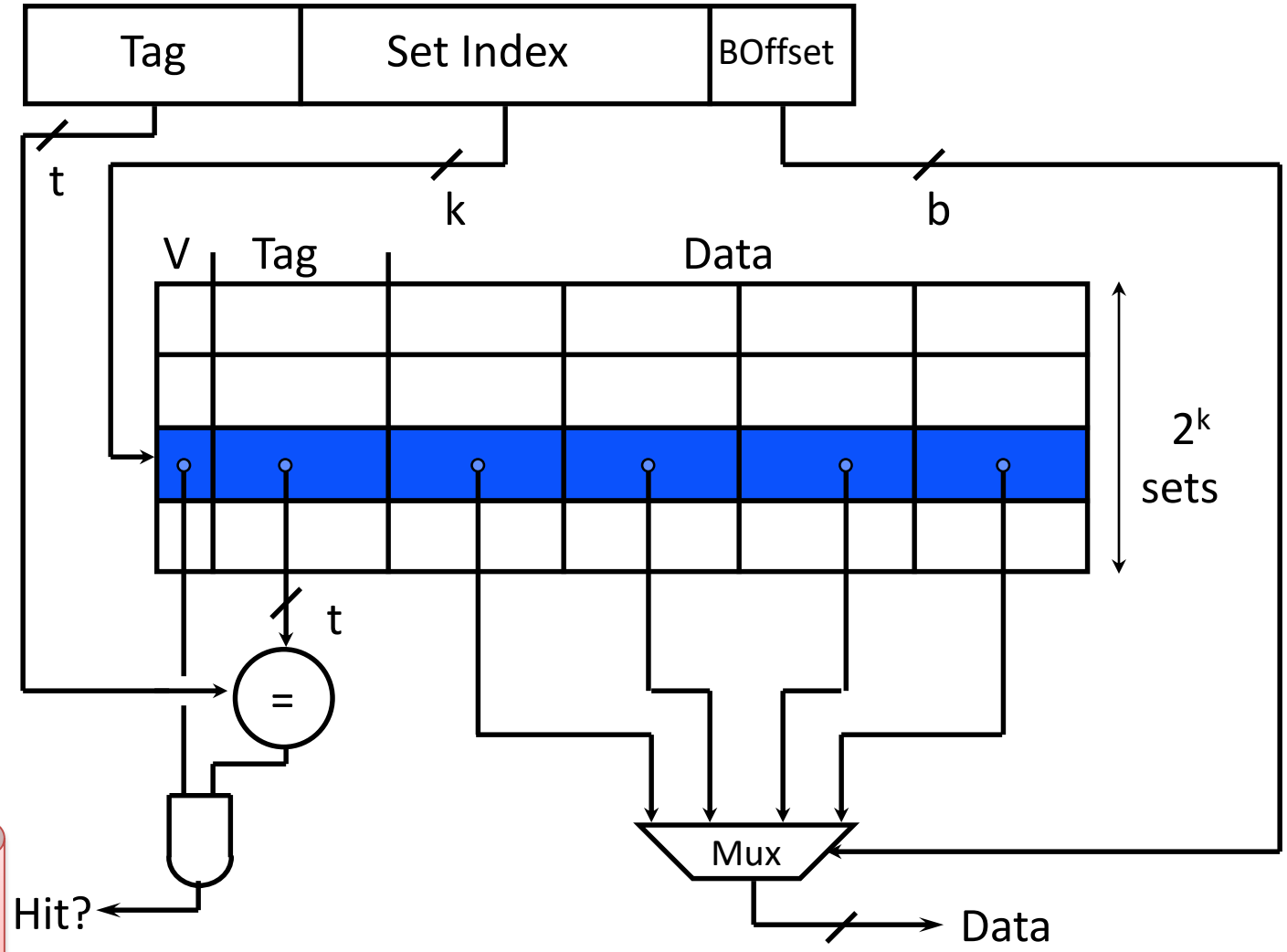
4-Way SA Cache Hardware (32-bit Memory)

- Assume: 32-bit memory address: 22-bit Tag, 8-bit set index, 2-bit Offset (each cache block is 4 Bytes/1 Word).
  Cache size: 4 ways * 2^8 (8-bit set index) = 1K blocks = 1K Words (4 KB)



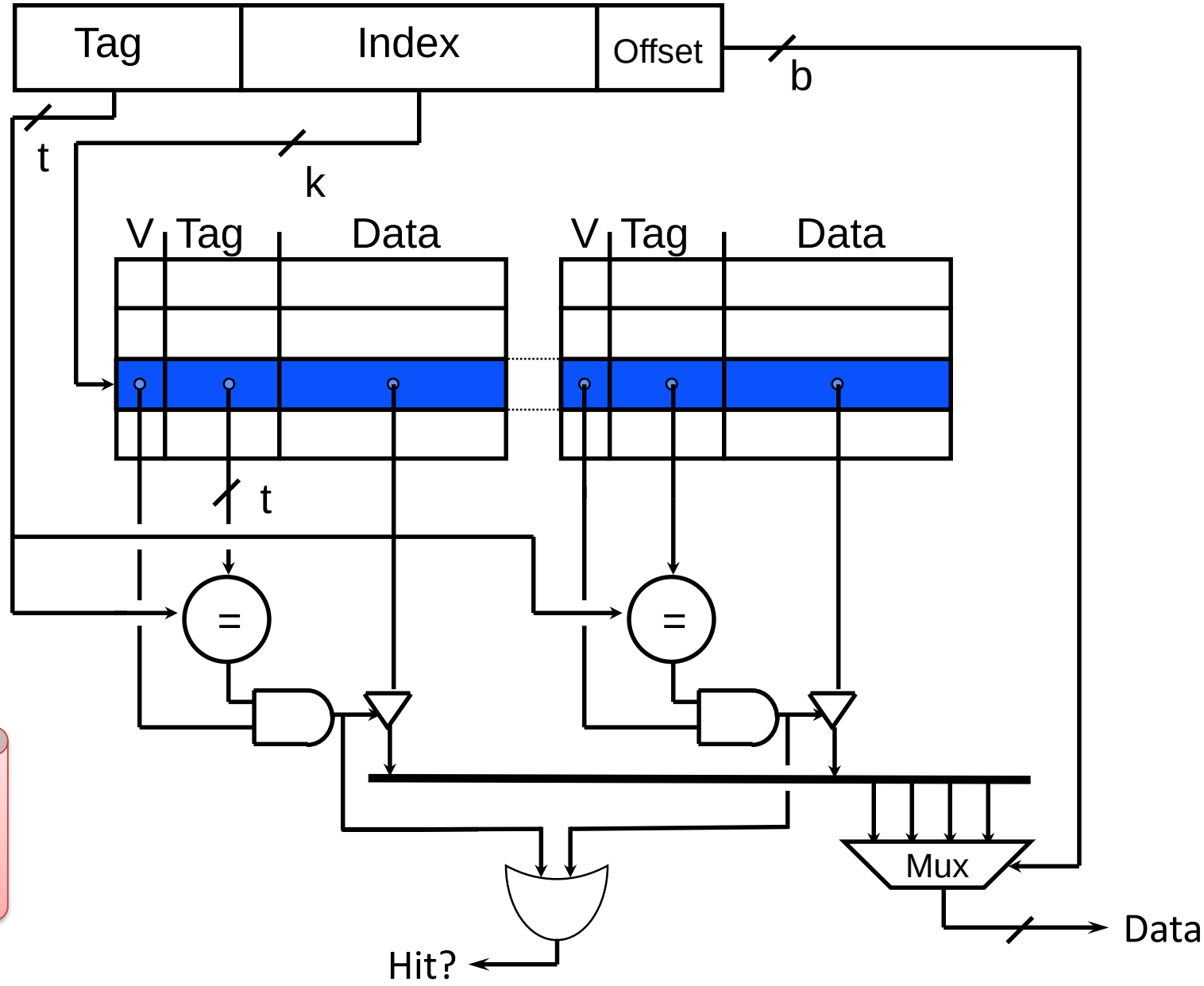
1. Set selection
2. Tag comparison
3. Data output

DM Cache HW Structure (Generic)



1. Set selection
2. Tag comparison
3. Data output

SA Cache HW Structure (Generic)



1. Set selection
2. Tag comparison
3. Data output

63

Alternative Cache Organizations: HW Costs

- Need to compare every tag in each cache set to the memory address; no need to compare tags in different cache sets
  - # HW comparators == # cache blocks in each cache set = # ways = associativity
- Direct Mapped: a memory block is mapped to one address in cache
  - # cache sets = # cache blocks; each cache set has 1 cache block
  - # HW comparators = 1
- Fully Associative: a memory block can be mapped to any address in cache
  - # cache sets = 1; it contains all cache blocks
  - # HW comparators = # cache blocks, plus a HW mux
- N-way Set Associative: a memory block can be mapped to one of N possible addresses in cache
  - # cache sets = # cache blocks / N; each cache set has N cache blocks
  - ***Fully Associative: N = total number of cache blocks***
  - ***Direct Mapped: N = 1***
  - N HW comparators, plus a HW mux

N-way SA Cache: Performance Costs

- Performance costs
  - N-way SA cache: N HW comparators make Hit/Miss decision; Mux selects a way out of N ways (the hit cache block); output data
  - DM cache: single HW comparator makes Hit/Miss decision; output data
    - Speculative execution optimization: output cache block data before Tag comparison makes Hit/Miss decision; execute optimistically assuming it is a hit; recover later if it turns out to be a miss; works well if hit rate is high
      - Instead of 1) Set selection; 2) Tag comparison; 3) Data output,
      - we have 1) Set selection; 2) (Data output & Tag comparison) in parallel
    - This optimization for DM cache does not work for N-way SA cache. Why?
      - For N-way SA cache, data is available only after Hit/Miss decision; if hit, the Mux selects a block from one of N ways
      - N possible hit cache blocks before Mux finishes selection
- But SA cache reduces miss rate by having more flexible cache block placement
  - When miss occurs, which block is selected for replacement?
  - Least Recently Used (LRU): one that has been unused the longest (principle of temporal locality)
- In summary: higher associativity → higher hit time & lower miss rate

Outline

- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

Handling Stores with Write-Through

- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: two policies

1) Write-Through Policy: write cache and write *through* the cache to memory
   - Every write updates both cache and memory
   - Optimization: insert a small write buffer between cache and memory to allow processor to continue once data in Buffer; memory is updated from write buffer in parallel to processor execution

Handling Stores with Write-Back

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

Write-Back Cache

- Store/cache hit, write data in cache *only* and set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - "Write-allocate" policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if it is dirty. Update cache with new block and clear dirty bit

Write-Through vs. Write-Back

- Write-Through:
  - Simpler control logic
  - More predictable timing simplifies processor control logic
  - Memory always has copy of all data in cache
- Write-Back
  - More complex control logic
  - More variable timing (0,1,2 memory accesses per cache access)
  - Sometimes cache has only copy of data that is not in memory
  - Reduces write traffic → write-back is the dominant approach in modern processors

Outline

- Memory Hierarchy and Latency
- Cache Principles
- Basic Cache Organization
- Different Kinds of Caches
- Write Back vs. Write Through
- Summary

Summary

- Cache – copy of data in lower level of memory hierarchy
- Principle of locality:
  - Program likely to access a relatively small range of memory addresses at any instant of time.
    - Temporal locality vs. spatial locality
- Cache organizations:
  - Direct Mapped: 1 block per set
  - N-way Set Associative: N blocks per set
  - Fully Associative: all blocks in 1 set
- Increasing associativity helps to reduce miss rate
  - N-way: N possible places in cache to hold a given memory block