ENGN2219/COMP6719
Computer Systems & Organization

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

Principle of Locality

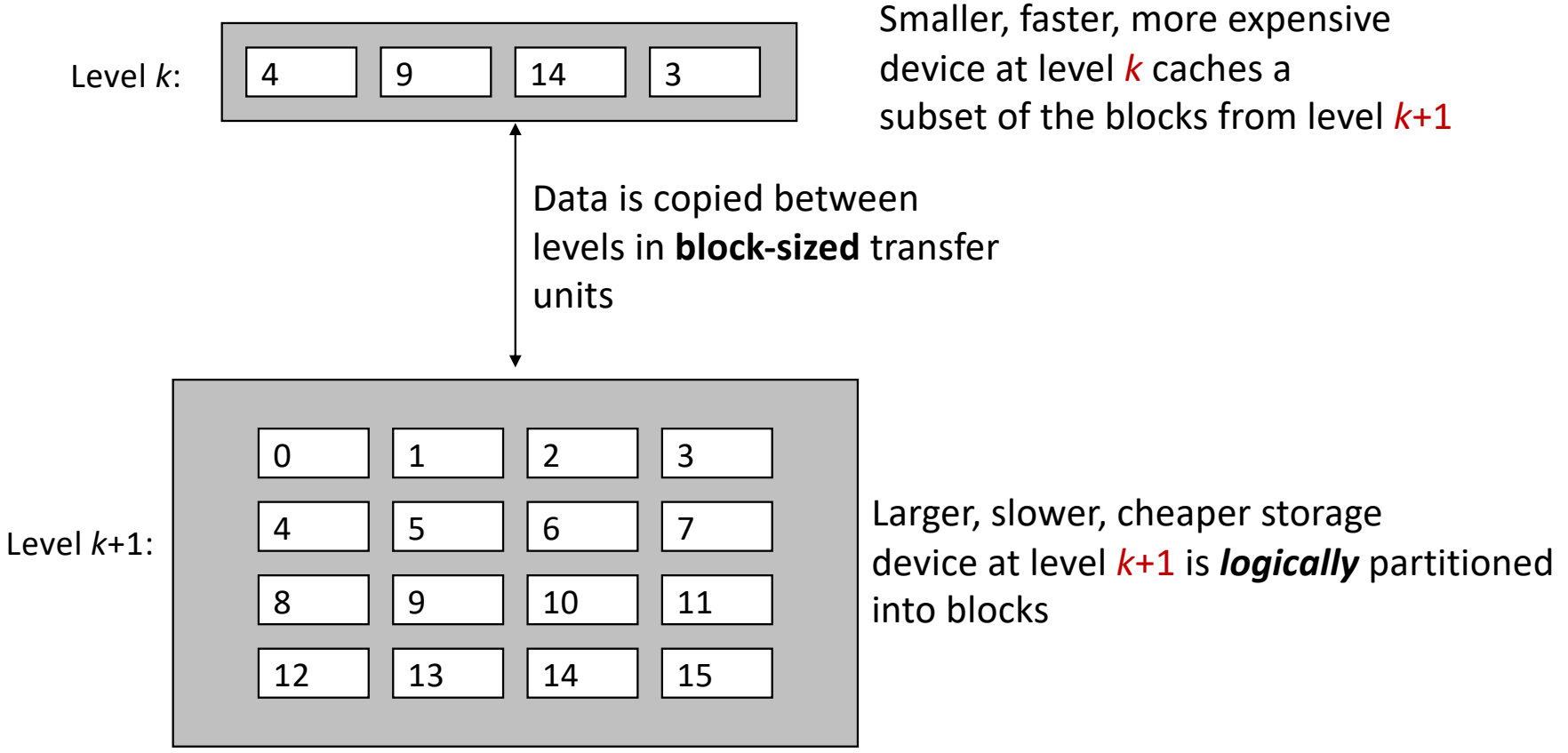
- Temporal Locality (locality in time)
  - If an item is accessed (referenced), it is likely to be referenced again soon
- Spatial Locality (locality in space)
  - If an item is referenced, items whose addresses are close by are likely to be referenced soon
- Well-written programs exhibit good locality
  - Programmers need to aim for high locality in programs
  - Sequential access patterns lead to better performance
- ***Today: How can we build hardware to exploit locality?***

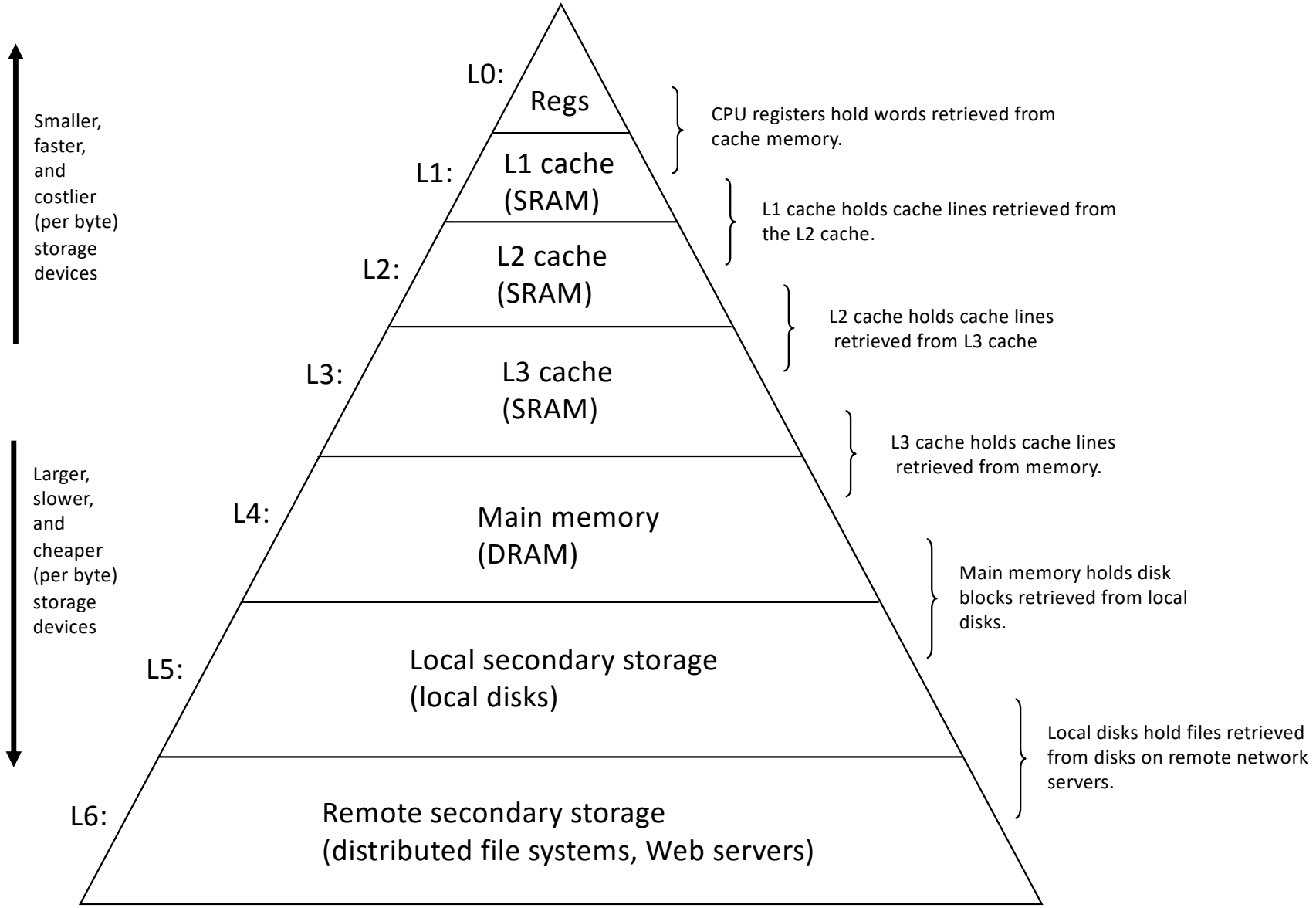
Real-Life Analogy

- In a library doing research on a subject
- *Temporal Locality:* If you brought a book to your desk to look up something, you are likely to look it up again
  - Keep the book on the desk instead of putting it back on the shelf
- *Spatial Locality:* Books on the same topic are put next to each other in a library
  - Bring several books on the subject to your desk to amortize the cost of walking to the shelf
- **Big Idea:** Desk serves as a small cache (*fast access*) for the large shelf (*slow access*)

Basic Idea of Caching

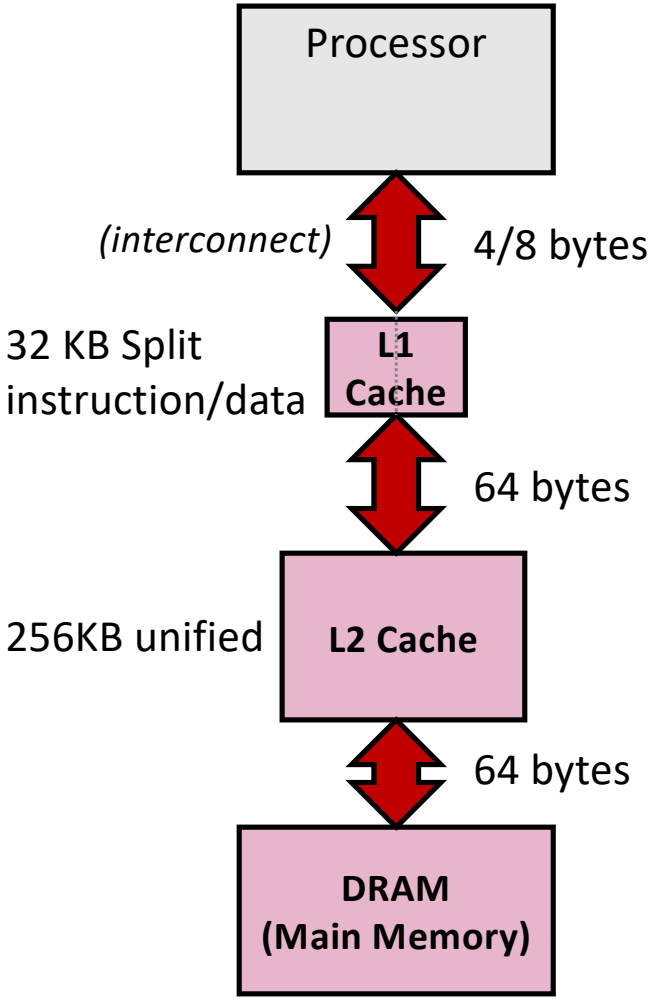
- **Cache:** A fast storage device that acts as a *staging area* for items stored in a larger, slower device




Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Transfer Granularity

- Data is always copied between level *k* and level *k+1* in block-sized units
- Block size is fixed between any pair of adjacent levels in the hierarchy
  - Other pairs of levels can have different granularity
- Typical granularity
  - 32– 64 bits between RF and L1-Cache
  - 64-bytes between L1 and L2
  - 64 bytes between L2 and L3 (not shown)
  - 64 bytes between L3 (not shown) and DRAM main memory



Hit Rate (Ratio)

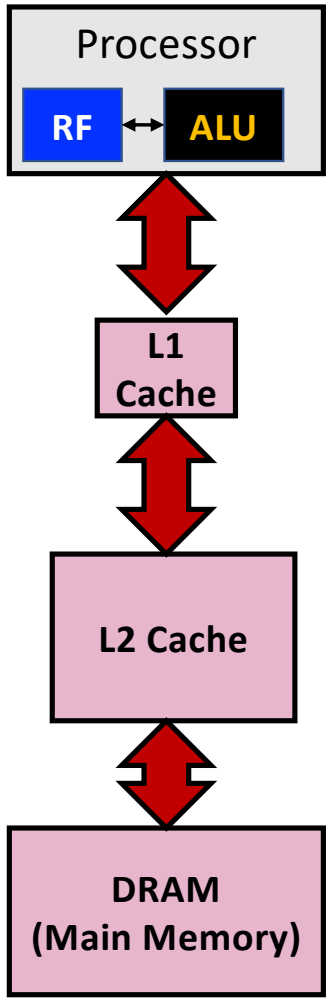
- Cache Hit
  - Program requests data word (d) at address **A** stored in level *k+1*
  - CPU first checks to see if the data word (d) is cached at level *k*
  - *If CPU finds d at level k, we call it a **cache hit***
  - *The CPU need not request d from level k+1*
- **Hit Rate**
  - Total # hits at level *k* `divided by` the total # accesses at level *k*

Miss Rate (Ratio)

- Cache Miss
  - Program requests data word (d) at address **A** stored in level *k+1*
  - CPU first checks to see if the data word (d) is cached at level *k*
  - *If CPU cannot find d at level k, we call it a **cache miss***
  - *The cache at level k needs to request d from level k+1*
- **Miss Rate:** Total # misses at level *k* `divided by` the total # accesses at level *k*
  - 1 – hit-rate

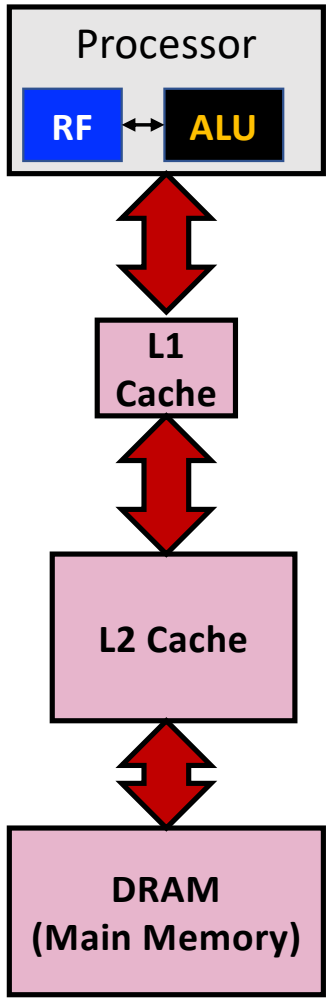
Hit Time

- The time required to:
  - Determine if a data word is cached at a particular level
  - Transfer the data word to the upper level
- Example: L1 Hit Time
  - Time it takes to determine if a data word is cached at Level 1
  - Time it takes to transfer the data word from L1-cache to the register file
  - Data movement from cache to RF includes both interconnect (wire) delay, read from SRAM cache, and write to the register



Miss Penalty

- In case of a cache miss at a particular level, the time required to
  - Retrieve the data word from the next level
  - Insert the data into the level that experienced the miss
- L1 Miss Penalty (assuming L2 Hit)
  - Time to find if data is in L2 (hit)
  - Transferring data from L2 to L1
- L1 Miss Penalty (assuming L2 Miss)
  - Time to find if data is in L2 (miss)
  - Transferring data from memory to L2
  - Transferring data from L2 to L1



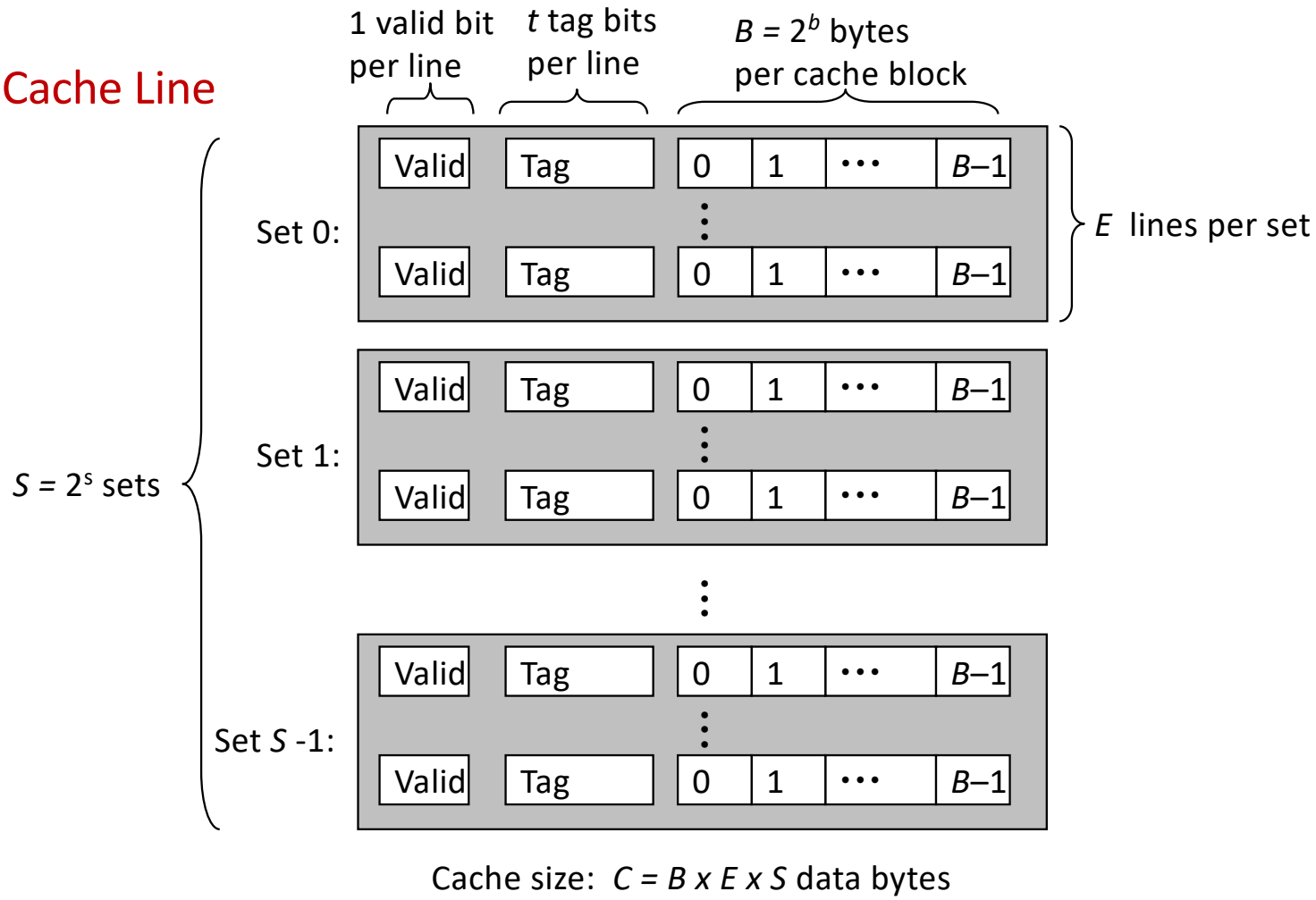
Generic Cache Organization

- Computer system with `m-bit` memory address
  - 2^m unique memory addresses
- A cache for such a system can only retain a small number of addresses
- When the CPU wants to read a word from address **A**, it first sends the request to cache memory
- Questions we will answer
  - How does the cache know whether it contains a copy of the word at address **A**?
  - How quickly can the cache determine whether it contains the copy of the word at address **A**?
  - How can we design the cache to maximize hit rate?

Generic Cache Organization

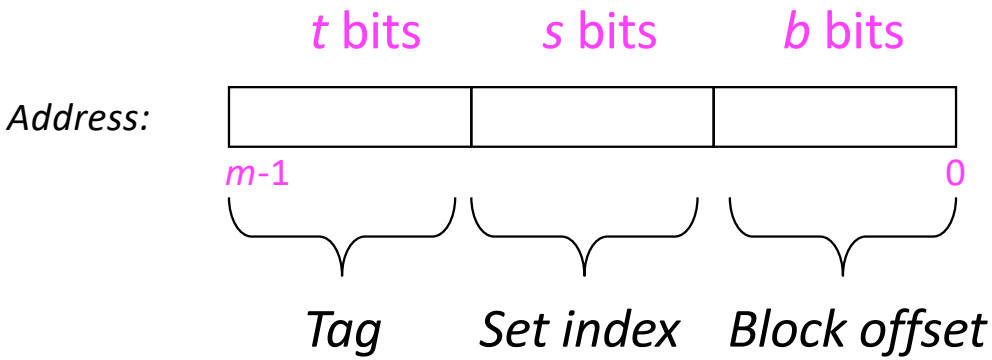
Note

Cache Block = Cache Line



Cache size: $C = B \times E \times S$ data bytes

Partitioning Address Bits



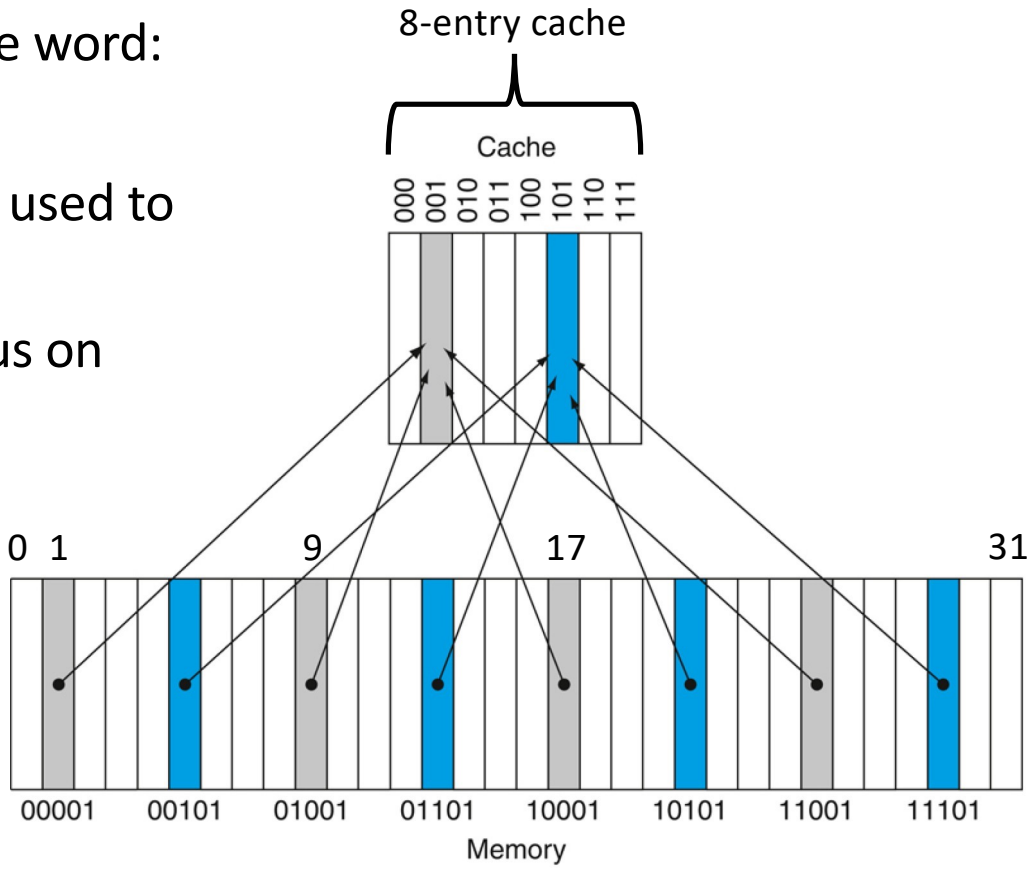
- Partition the `m` address bits in address **A** into three fields
  - The `s` set index bits form an index into the array of `S` sets (0, 1, 2, ...)
  - The `t` tag bits in **A** determine which line in set contains the word
  - The `b` block offset bits give the offset of the word in the `B–byte` data block
- A line in the set contains the word if and only if
  - The *valid bit* is set
  - The *tag bits* in the line match the tag bits in the address **A**

Direct-Mapped Cache

- A cache with exactly one line per set (E = 1)
- Each line in the address space can reside at only one location in the direct-mapped cache
- Any cache needs to perform three actions
  - Set selection
  - Line matching
  - Word extraction
- Set selection in direct-mapped cache
  - Block address `modulo` # blocks in cache
- Line matching
  - Compare the tag bits in the line to the tag bits in the address
- Word extraction: use the block offset bits to extract the word

Direct-Mapped Cache

- An address `X` maps to direct-mapped cache word:
  - `X modulo 8`
- Low order 3 bits are used to index the cache
- To simplify, let's focus on block accesses



- Memory is partitioned into several fixed-sized blocks

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110		10110 mod 8 = 110
26	11010		11010 mod 8 = 010
22	10110		10110 mod 8 = 110
26	11010		11010 mod 8 = 010
16	10000		10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Initial State of the Cache

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010		11010 mod 8 = 010
22	10110		10110 mod 8 = 110
26	11010		11010 mod 8 = 010
16	10000		10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Miss 10110

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110		10110 mod 8 = 110
26	11010		11010 mod 8 = 010
16	10000		10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Miss 11010

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010		11010 mod 8 = 010
16	10000		10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Hit 10110

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000		10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Hit 11010

Index	V	Tag	Data
000	0		
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000	miss	10000 mod 8 = 000
3	00011		00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Miss 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	0		
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000	miss	10000 mod 8 = 000
3	00011	miss	00011 mod 8 = 011
16	10000		10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Miss 00011

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000	miss	10000 mod 8 = 000
3	00011	miss	00011 mod 8 = 011
16	10000	hit	10000 mod 8 = 000
18	10010		10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Hit 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	11	Mem[11010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000	miss	10000 mod 8 = 000
3	00011	miss	00011 mod 8 = 011
16	10000	hit	10000 mod 8 = 000
18	10010	miss	10010 mod 8 = 010
16	10000		10000 mod 8 = 000

Miss 10010

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

- Due to a conflict, we must remove a valid block from the cache

Accessing a Cache

Address (Decimel)	Binary Address	Hit/Miss in Cache	Assigned Cache Block
22	10110	miss	10110 mod 8 = 110
26	11010	miss	11010 mod 8 = 010
22	10110	hit	10110 mod 8 = 110
26	11010	hit	11010 mod 8 = 010
16	10000	miss	10000 mod 8 = 000
3	00011	miss	00011 mod 8 = 011
16	10000	hit	10000 mod 8 = 000
18	10010	miss	10010 mod 8 = 010
16	10000	hit	10000 mod 8 = 000

Hit 10000

Index	V	Tag	Data
000	1	10	Mem[10000]
001	0		
010	1	10	Mem[10010]
011	1	00	Mem[00011]
100	0		
101	0		
110	1	10	Mem[10110]
111	0		

Line Replacement

- One a miss, the cache needs to retrieve the block from the next level
- If there are valid blocks in the cache, then one of them must be evicted (in case of conflict) to make room for the new block
  - Which of the many possible blocks to evict is decided by the cache replacement policy
- For a direct-mapped cache, the replacement policy is trivial
  - Replace the current block (line) with the newly inserted block (line)

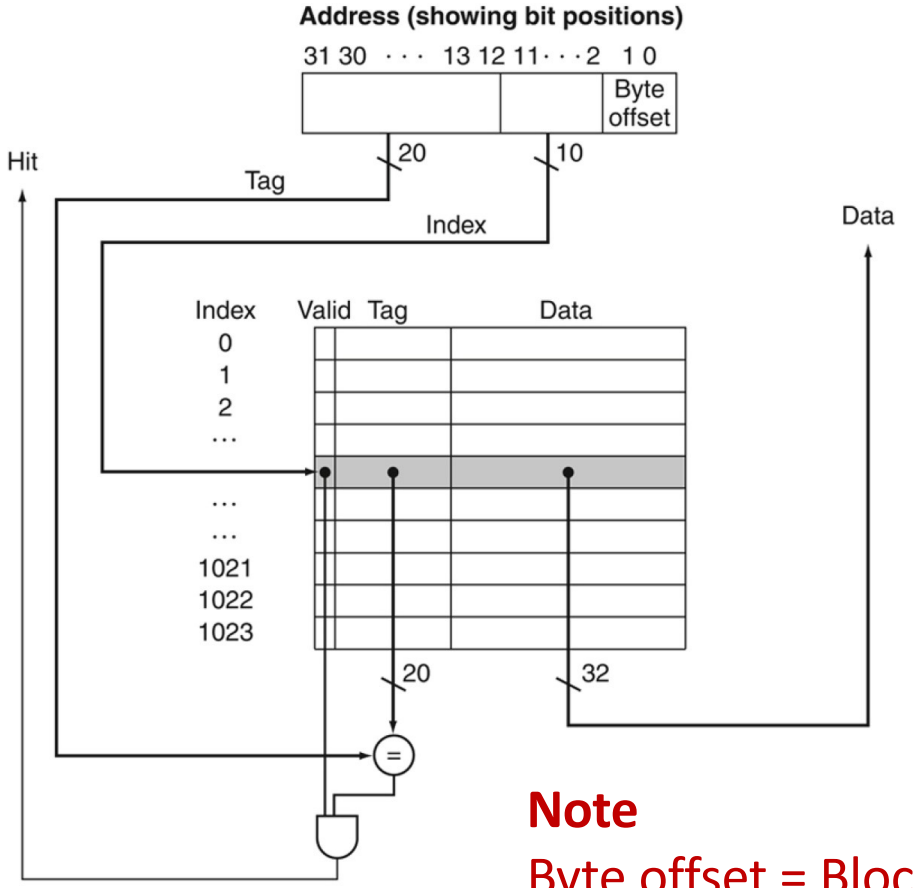
Example: Direct Mapped Cache
Block size = 1 word = 4 Bytes

Size of cache in bits

- 1024 words
- 4 KB (tag/valid not counted)
- Index = 10 bits
- Tag = 32 – 10 – 2 = 20 bits

Operation

- `valid` bit is 1
- `tag bits` match
- cache hit!



Note

Byte offset = Block offset

Exercise

How many total bits are required for a direct-mapped cache (including valid and tag bits) with 16 KB of data and 4-word blocks, assuming a 32-bit address? The word size is 4 bytes.

Block size = 16 Bytes → 4 bits *for block offset*
Blocks = 16 KB/16 = 1024 → 10 bits *for set index*
bits for tag = 32 – 10 – 4 = 18 bits *for tag*
bits for meta-data = 1 bit *for valid*

total bits = 2^{10} X (4 X 32 + 19) = 147 Kbits

Exercise

Consider a cache with 64 blocks and a block size of 16 bytes. To which block number does byte address 1200 maps?

Address of block is given by byte address divided by bytes per block

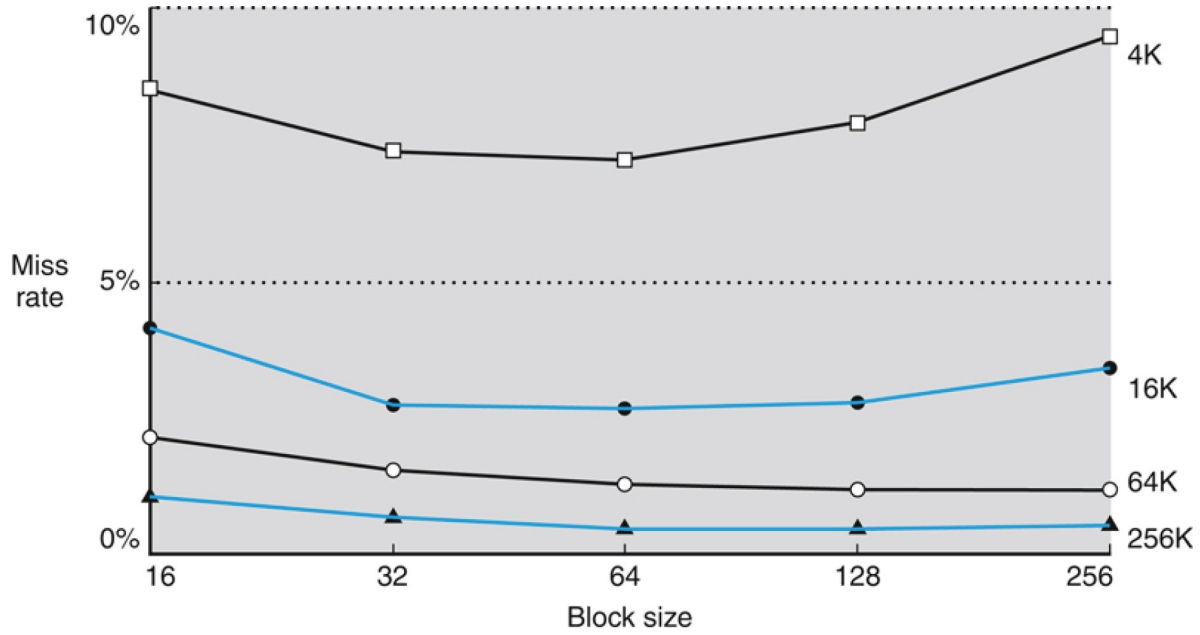
- = 1200/16 = 75 (interpretation: every 16 bytes, a new block starts)

The (cache) block is given by: Block address `modulo` (# blocks in cache)

- = 75 modulo 64 = 11

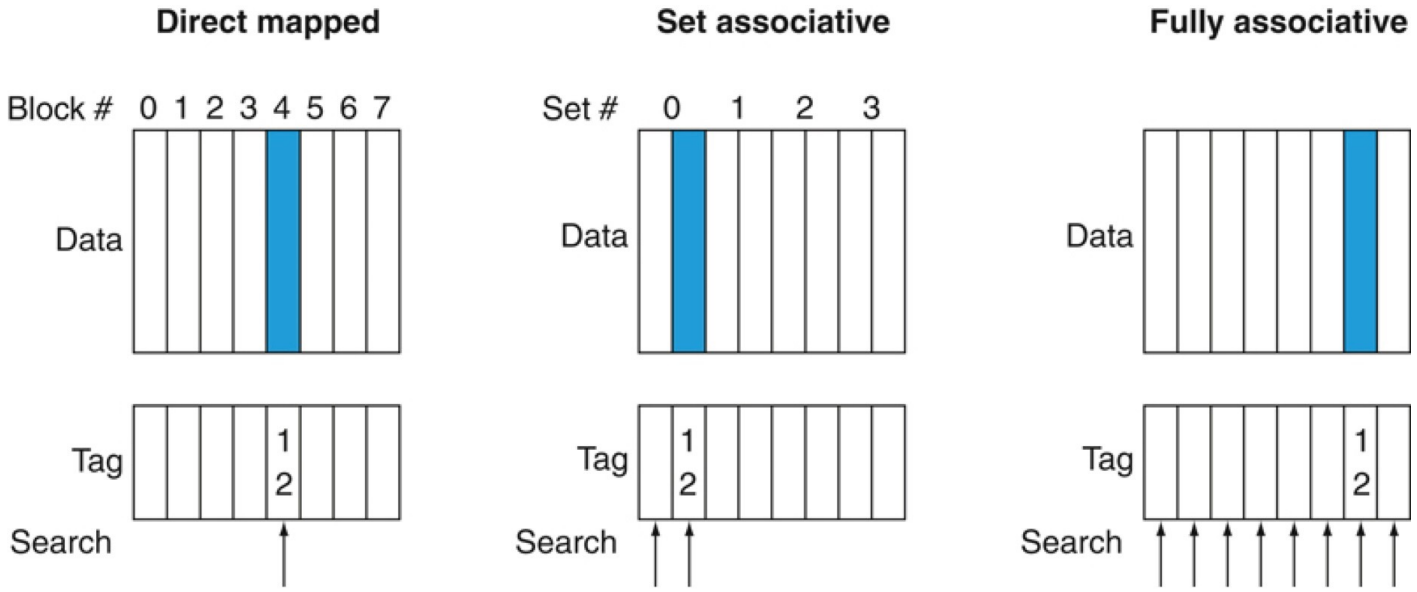
Miss Rate vs. Block Size

- When the block size becomes a significant fraction of the cache size, then two aspects become critical
  - There is a small # blocks in the cache
  - There is a greater competition for the small # blocks



Flexible Block Placement

- **Direct-mapped:** *Each block can go to only location only*
  - Leads to a lot of conflict misses
- **E-way set associative:** *Each block can go to one of* E *locations inside a set (also called* ***n-way set-associative****)*
- **Fully associative:** *Each block can go to any location inside the cache*



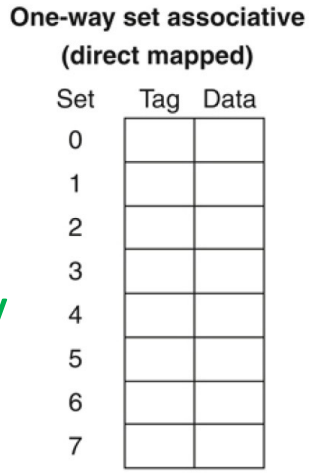
Finding a Block with Associativity

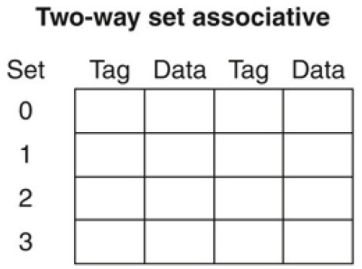
- Set selection in E-way set-associate cache
  - Block address `modulo` # sets in cache
- Line matching
  - Compare the tag bits of each block in the set to the tag bits in the address
- Word extraction
  - Use the block offset bits to extract the word

Every Cache is a Set-Associative Cache!

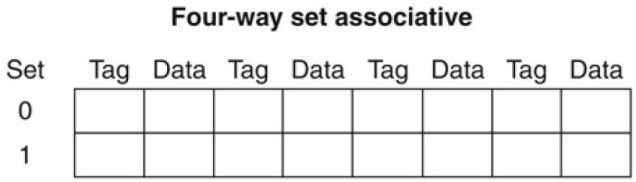
For a cache with 8 entries

8 sets, 1 way



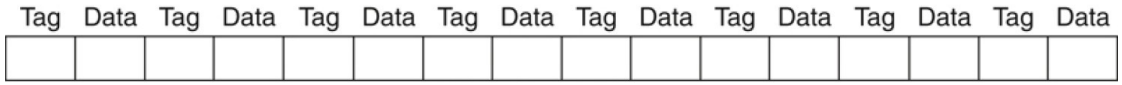


4 sets, 2 way



2 sets, 4 way

1 set, 8 way



Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0						
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully-associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - **Direct-mapped**
  - 2-way set associative
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Direct-mapped

Address mappings
0→0
6→2
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Common Replacement Policies

- Set associative caches need a more elaborate replacement **policy**
- Random
  - *Pick a random block to make room for the newly fetched block*
- Least Recently Used (LRU)
  - *Pick a line that was last accessed the furthest in the past*
  - References: B, C, C, C, B, B, A → Replace C with A
  - C is the least recently used, i.e., oldest access time
  - B has better temporal locality
- Least Frequently Used (LFU)
  - *Pick a line that has been referenced the fewest times over some past time window*
- Most Recently Used (MRU)
  - *Pick a line that has been referenced the latest in the past*

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - **2-way set associative**
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Address mappings
0→0
6→0
8→0

2-way set associative, LRU replacement

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - **2-way set associative**
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings
0→0
6→0
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - **2-way set associative**
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

2-way set associative, LRU replacement

Address mappings
0→0
6→0
8→0

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0					
8	0					

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - **2-way set associative**
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Address mappings
0→0
6→0
8→0

2-way set associative, LRU replacement

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0					

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - **2-way set associative**
  - Fully associative
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Address mappings
0→0
6→0
8→0

2-way set associative, LRU replacement

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - 2-way set associative
  - **Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8						
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - 2-way set associative
  - **Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0						
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - 2-way set associative
  - **Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6						
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - 2-way set associative
  - **Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8						

Associativity Example

- Let us compare different caches with four blocks and one word per block
  - Direct-mapped
  - 2-way set associative
  - **Fully associative**
- Address references: 0, 8, 0, 6, 8

Key:

Red: miss
Green: hit
Blue: untouched

Fully associative

Address mappings
0→anywhere
6→anywhere
8→anywhere

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

Performance Comparison

- In terms of speed (hit time)
  - Direct mapped cache is the fastest
    - Only one place to look for the requested block
  - Fully-associative cache is the slowest
    - Many places to search for the requested block
  - n-way set-associative cache is a compromise
- In terms of miss rate
  - Direct mapped cache is the worst
    - Many conflicts due to lack of flexible placement
  - Fully-associative cache is the best
    - Flexibility is very high
  - n-way set-associative cache is a compromise

Common Addressing Scheme

tag	index	block offset

- 2-way set-associative
  - 64-byte block/line size
  - 2048 blocks (1024 sets)
  - 32-bit address
  - How many bits are needed for the tag, index, and block offset?

tag (16)	index (10)	block offset (6)

Practice

A cache has 4096 blocks and a 16-byte block size. Assuming 32-bit addresses, find the total number of sets and the total number of tag bits for caches that are (1) direct-mapped (2) 2-way set associative (3) 4-way set-associative (4) fully associative.

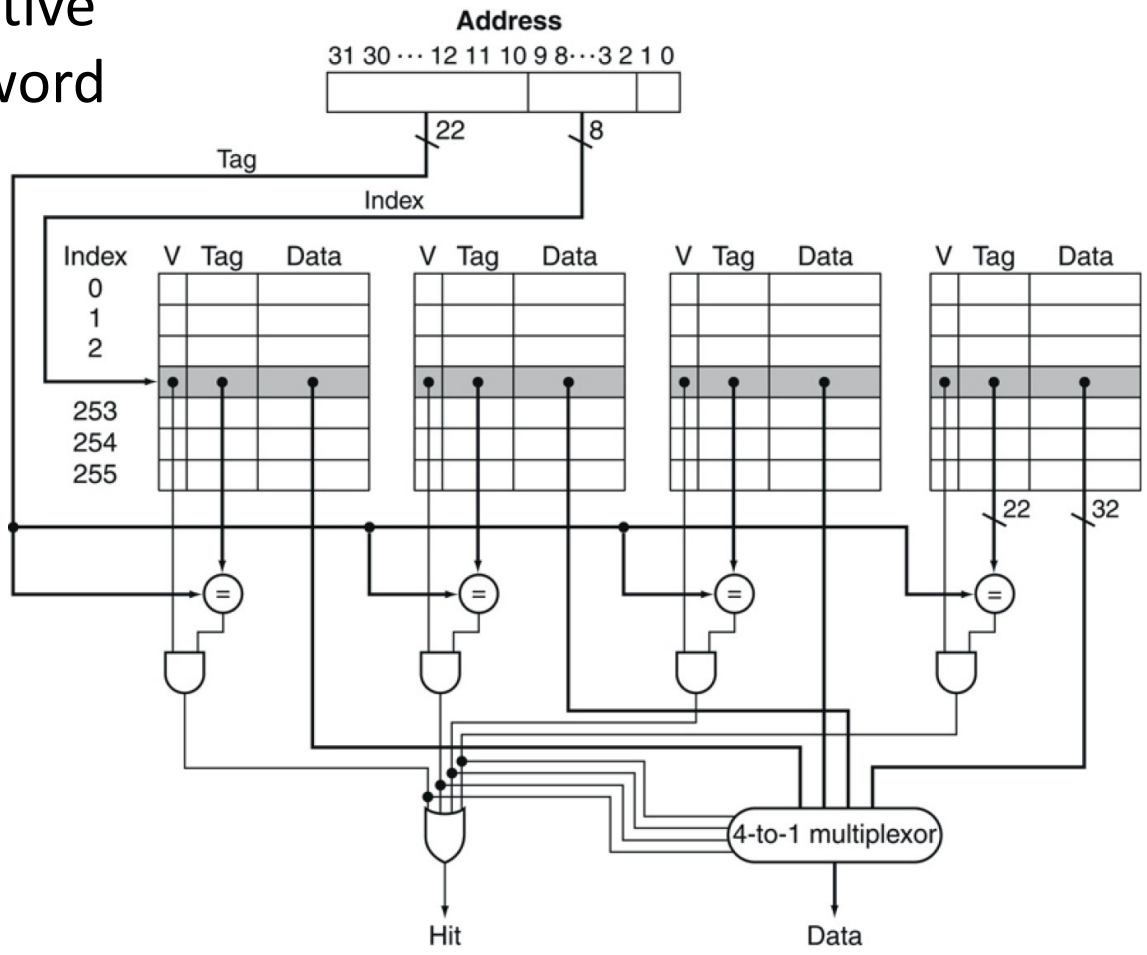
bits for index + tag = 32 – 4 = 28 (4-word block means 16 addressable bytes in each block)
sets in direct-mapped cache = 4096 = 12-bit index and 16-bit tags
tag bits = 16 * 4096 = 66 K tag bits

Each degree of associativity decreases the # sets by 2 and decreases the # bits to index the cache by 1 and increases the tag bits by 1. A fully-associative cache has one set.

- 2-way cache: 2048 sets, tag bits = (28 – 11) * 2 * 2048 = 70 Kbits
- 4-way cache: 1024 sets, tag bits = (28 – 10) * 4 * 1024 = 74 Kbits
- Fully associative: 28 * 4096 * 1 = 115 Kbits

Architecture: Set-Associative Cache

4-way set-associative
Block size = one word
= 4 Bytes



Area Comparison

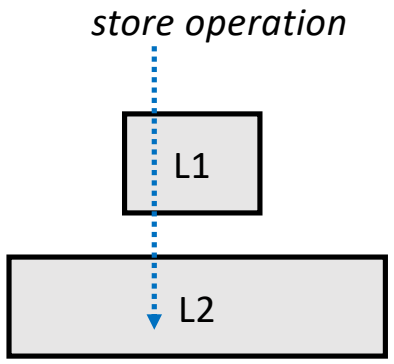
- Today's SRAM caches are on the same die as the processor
  - We call these on-chip or on-die caches
  - Reducing the area complexity is important so we can have a cache hierarchy with high capacity
- In terms of die area (logic complexity)
  - Direct mapped cache takes up the least die area as there is only one comparator and no multiplexer
  - Increasing associativity increases the # comparators and the width of the multiplexer

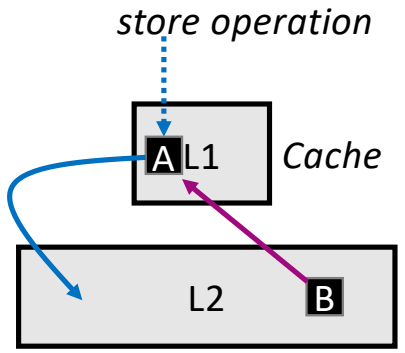
Issues with Writes

- Reads (loads) are relatively straightforward
- Writes (stores) open up a number of interesting problems
- Write hit
  - On a write to a cache line (write hit), what should we do about updating the copy of the cache line in the next lower level?
- Write miss
  - To allocate or not to allocate a block on a write miss

Write-Through vs. Write-Back

- Write-through
  - Immediately write the updated cache block to the next lower level
  - Simple to implement
  - Write traffic on the bus with every write
- Write-back
  - On a write hit, update the block in the cache only
  - Defer the write to the next lower level as much as possible
  - Write the modified block (A) to the next level only when it is evicted from the cache by the replacement algorithm
  - Need an additional meta-data bit: *dirty bit*





Allocating Blocks on Writes

- Write-Allocate
  - Load the corresponding block from the next lower level and then update the block
  - Exploits spatial locaity
  - Every miss results in a block transfer from the next lower level
- No-Write-Allocate
  - Bypass the cache and write the word directly to the next lower level
- Popular choices
  - Write-through + no-write-allocate
  - Write-back + write-allocate

Writeback Buffer

- On a cache miss, a write-back cache needs to perform two main tasks involving data movement
  - Write the replacement candidate to the next lower level of the memory hierarchy
  - Bring the cache block from the next lower level and insert it in the level that initiated the miss
- Which one should the cache do first?
- There is only one answer as the cache cannot overwrite the modified block (*write the replacement candidate first*)
  - **Optimization:** Prioritize returning data word to the processor
  - Move the evicted block to a special *writeback buffer* and first bring the data from the next level into the cache

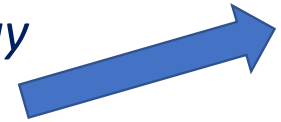
The 3C Model

- Compulsory (Cold) Misses
  - These misses are caused by the first access to a block that has never been in the cache
- Capacity Misses
  - These are cache misses caused when the cache cannot contain all the blocks needed during program execution
  - One way to think: *if we end up replacing blocks with a fully-associative cache*
- Conflict (Collision) Misses
  - Misses that occur when multiple blocks compete for the same set
  - One way to think: *eliminated by a fully-associative cache of the same size*

Software Interaction with Caches

```
struct point_vector {
        int point_x;
        int point_y;
};
struct point_vector p[1024]
…
…
while (i < 1024) {
        p[i].point_x *= 10;
}
```

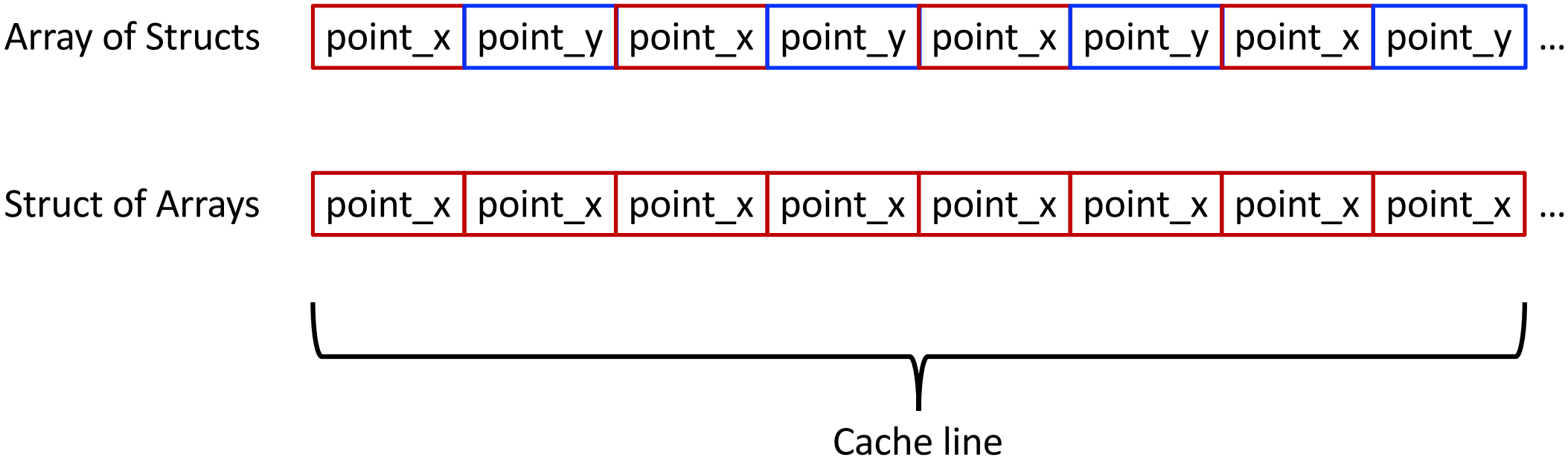
Question: *Can we transform the array of structs (AoS) representation into a format that results in better locality of referece?*



```
struct point_vector {
        int point_x[1024];
        int point_y[1024];
};
struct point_vector p;
…
…
while (i < 1024) {
        p.point_x[i] *= 10;
}
```

Struct of Arrays (SoA)

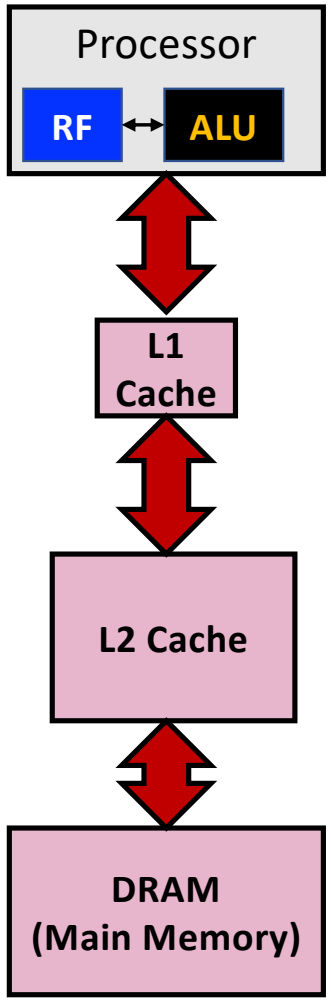
SoA vs AoS



If we are only interested in manipulating all points of a single type (x in our case) then the SoA layout can bring from memory and locate more points in a single cache line

Exercise (Offline)

- Find the average memory access time given
  - L1 Hit Time = 1 Cycle
  - L1 Hit Rate = 80%
  - L2 Hit Time = 10 cycles
  - L2 Hit Rate = 50%
  - L2 Miss Penalty = 200 cycles



Cache Size vs. Associativity

- If we use a 2-way set associative cache w/t eight blocks (instead of four blocks), and compare its miss rate to a fully-associative cache with four blocks, we find that
  - There are no replacements in the 2-way set associative cache with eight blocks
  - It's miss rate is the same as the fully associative cache
- If we repeat the exercise with 16 blocks for all three caches, we find that
  - All three caches will have the same miss rate

- **Bottomline**
  - Cache size and associativity both determine cache performance