# Memory & Cache

Zhaoguo Wang

# Question

For instruction `movq (%rax), %rbx`, how many memory accesses need to be made?

# Question

For instruction `movq (%rax), %rbx`, how many memory accesses need to be made?

Can we avoid all memory accesses?

# Principle of locality

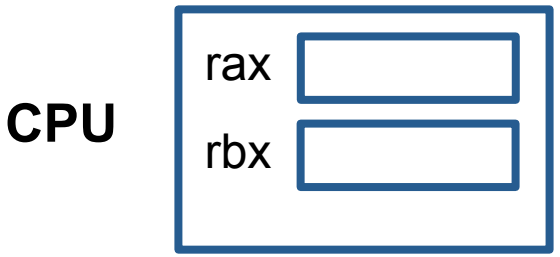
Temporal locality

- If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.

Spatial locality

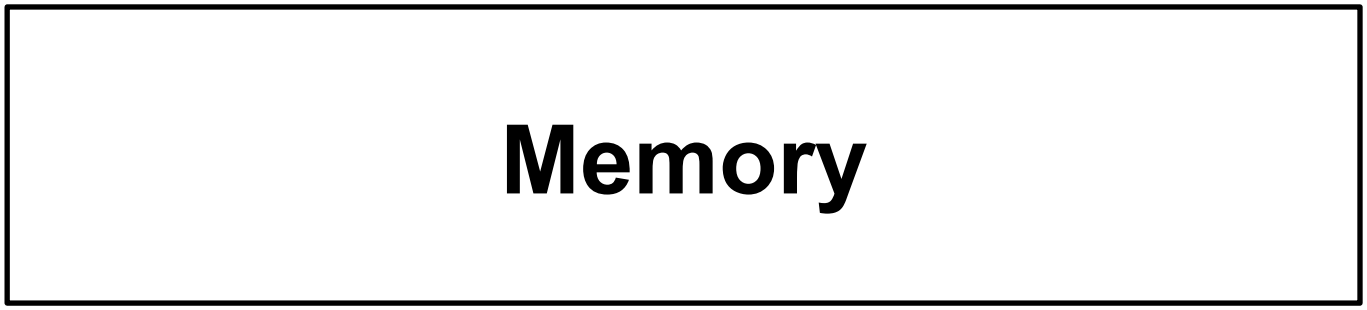
- If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
- Sequential locality
  - occurs when data elements are arranged and accessed linearly

# Basic idea – caching

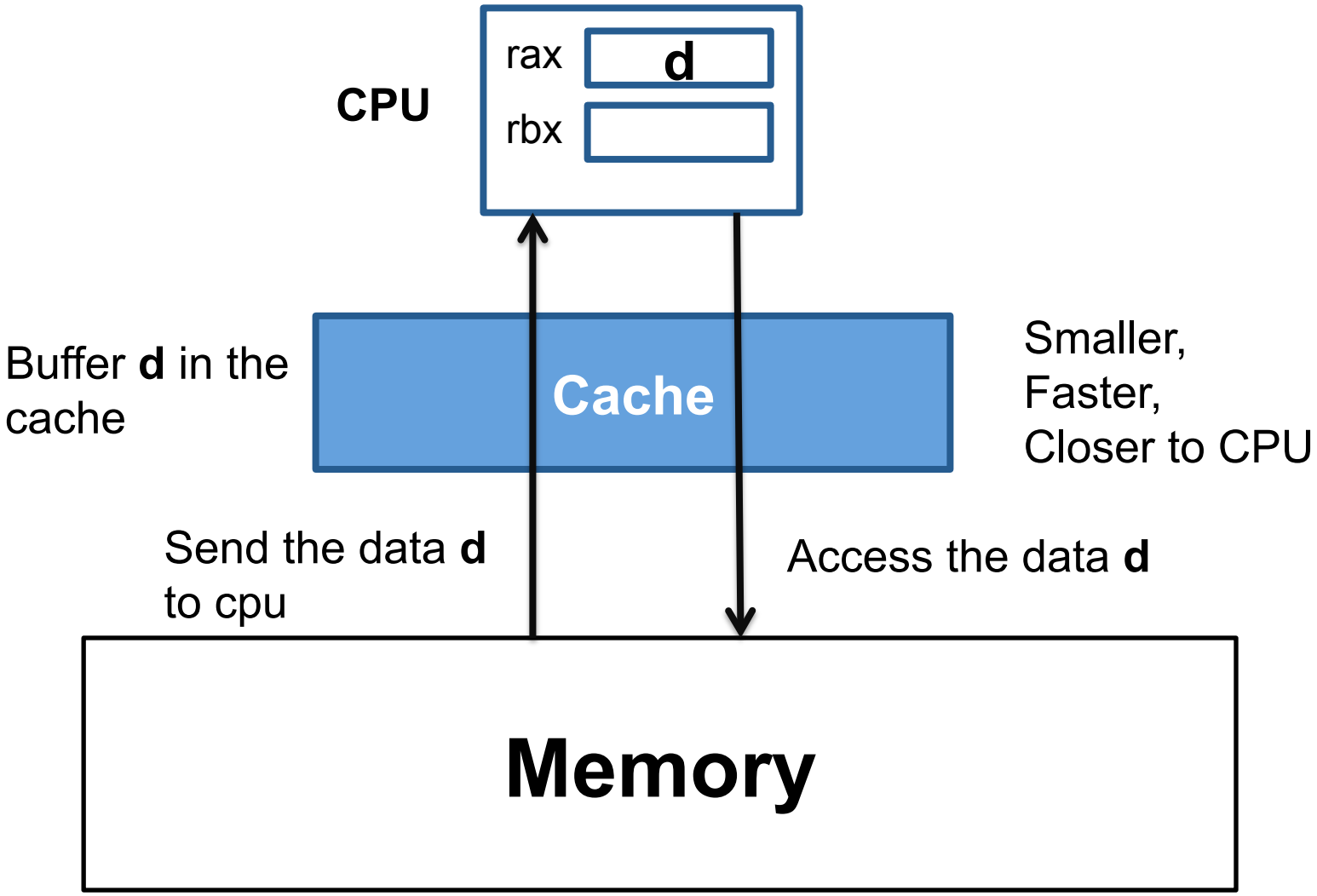




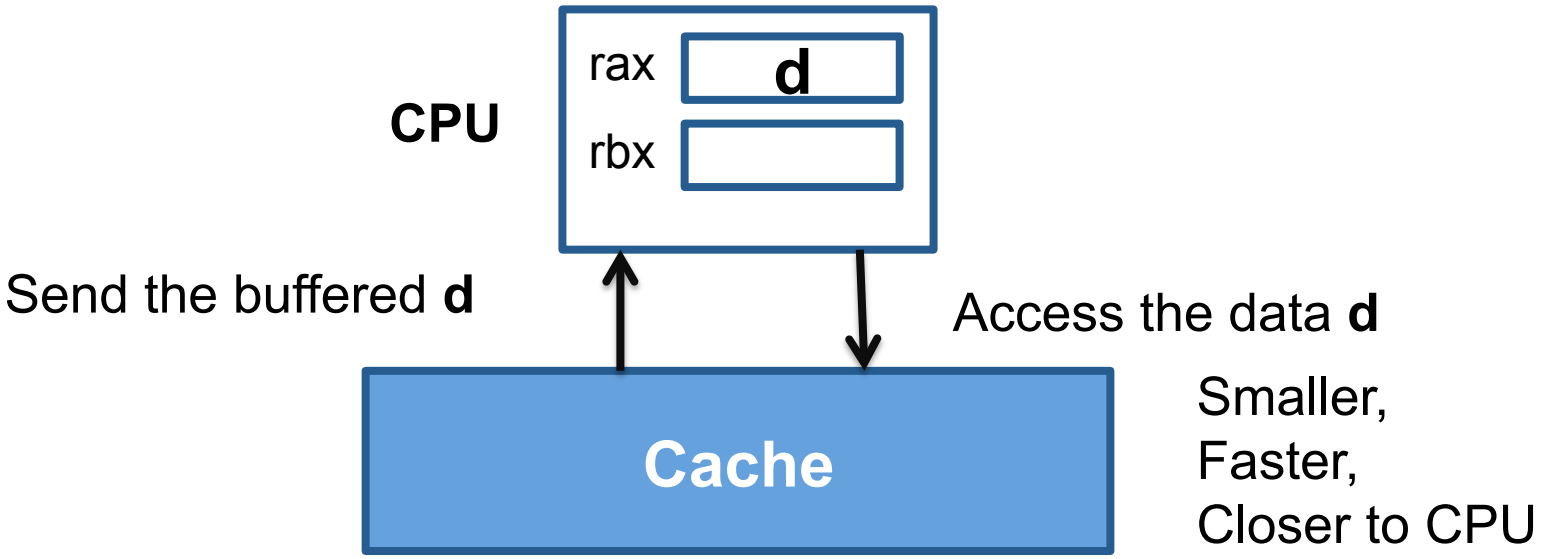
Smaller,
Faster,
Closer to CPU



# Basic idea – caching

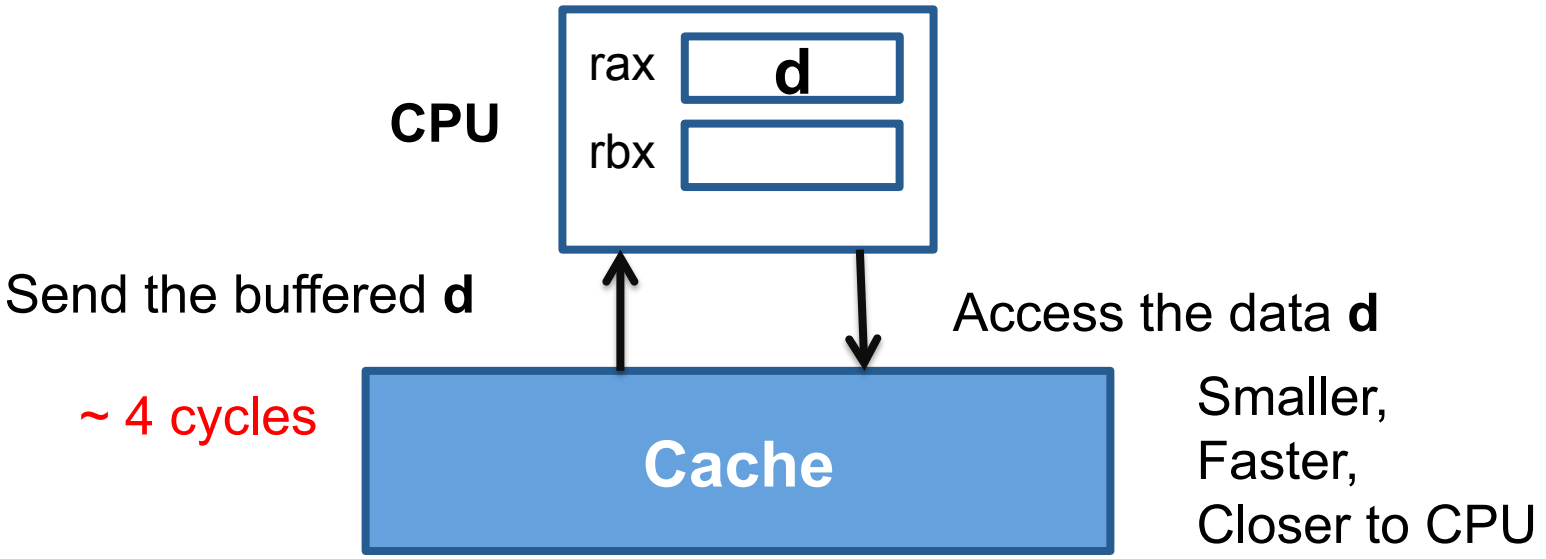


# Basic idea – caching




**Memory**

# Basic idea – caching




100 ~ 200 cycles

**Memory**

# Intuitive implementation

Caching at byte granularity:

- Search the cache for each byte access
  - movq (%rax), %rbx causes 8 times check
- High cost to maintain the address information

| PA | 1 byte |
|---|---|
| 0x100 | … |
| 0x101 | … |
| 0x102 | … |
| 0x103 | … |

# Caching at block granularity

Observation

- Spatial locality

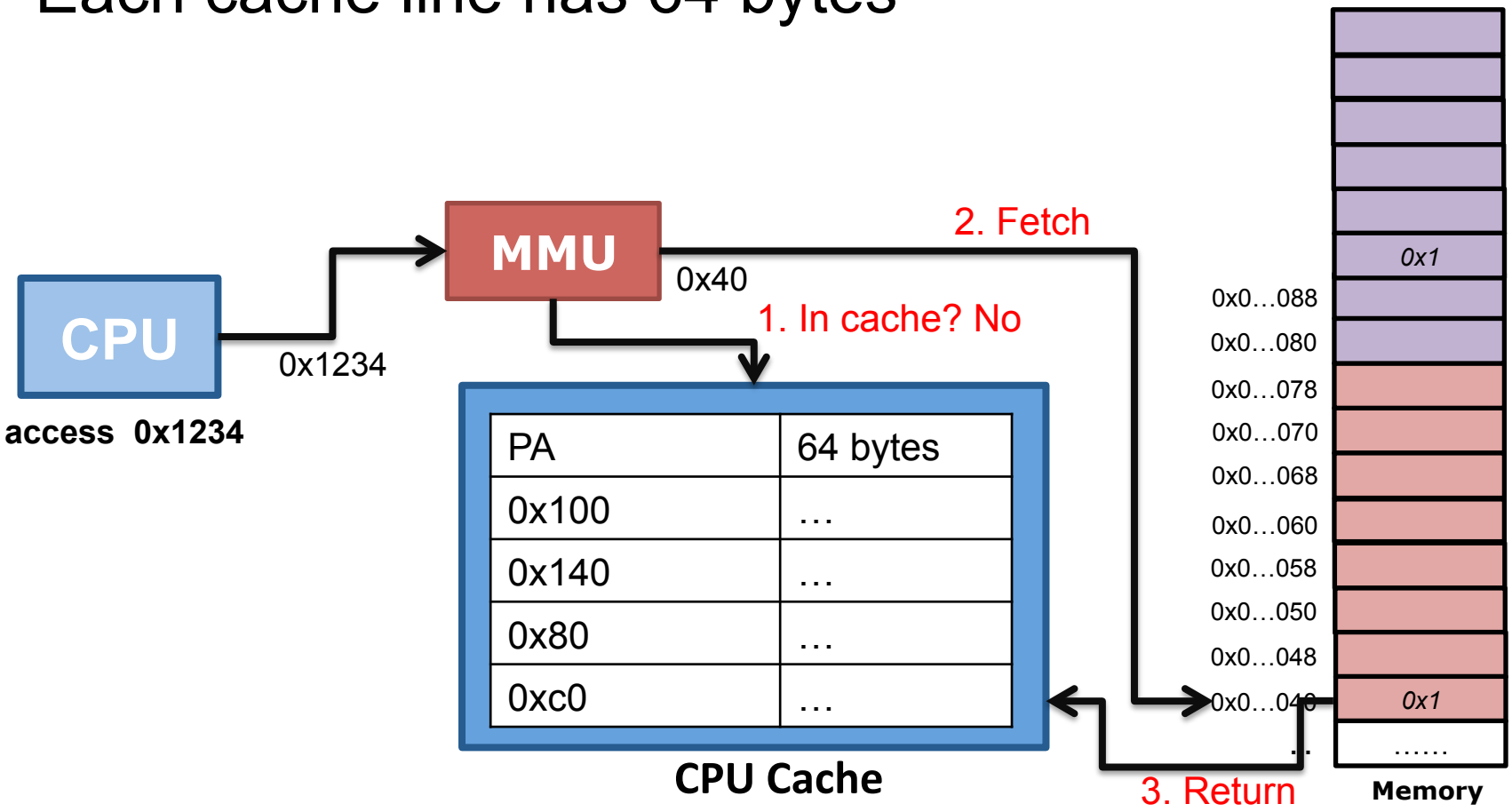
Solution

- Caching at block granularity
- Each block is called cache line, which has 64 bytes

# Direct-mapped cache

Caching at block granularity

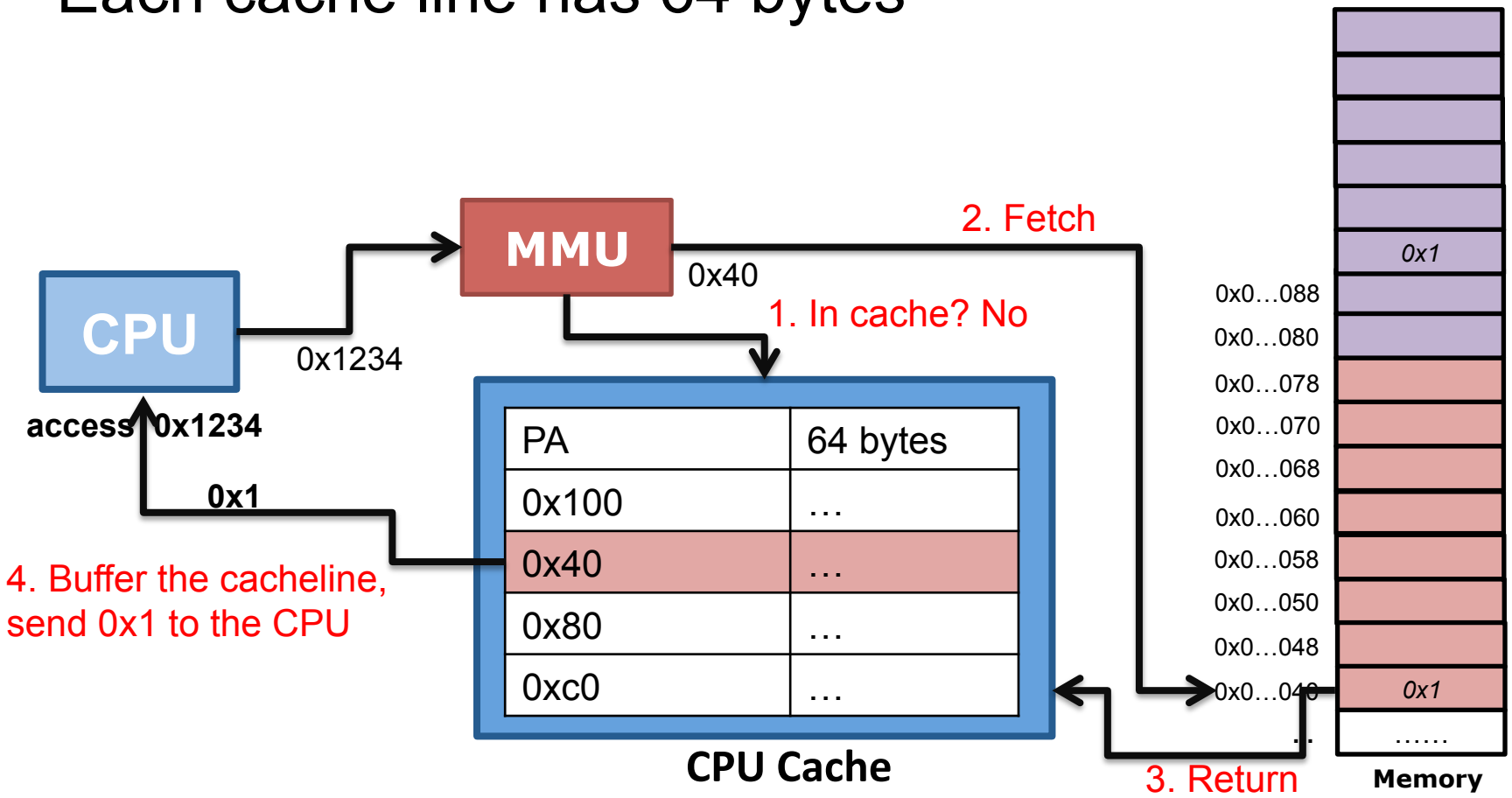
- Each cache line has 64 bytes



# Direct-mapped cache

Caching at block granularity

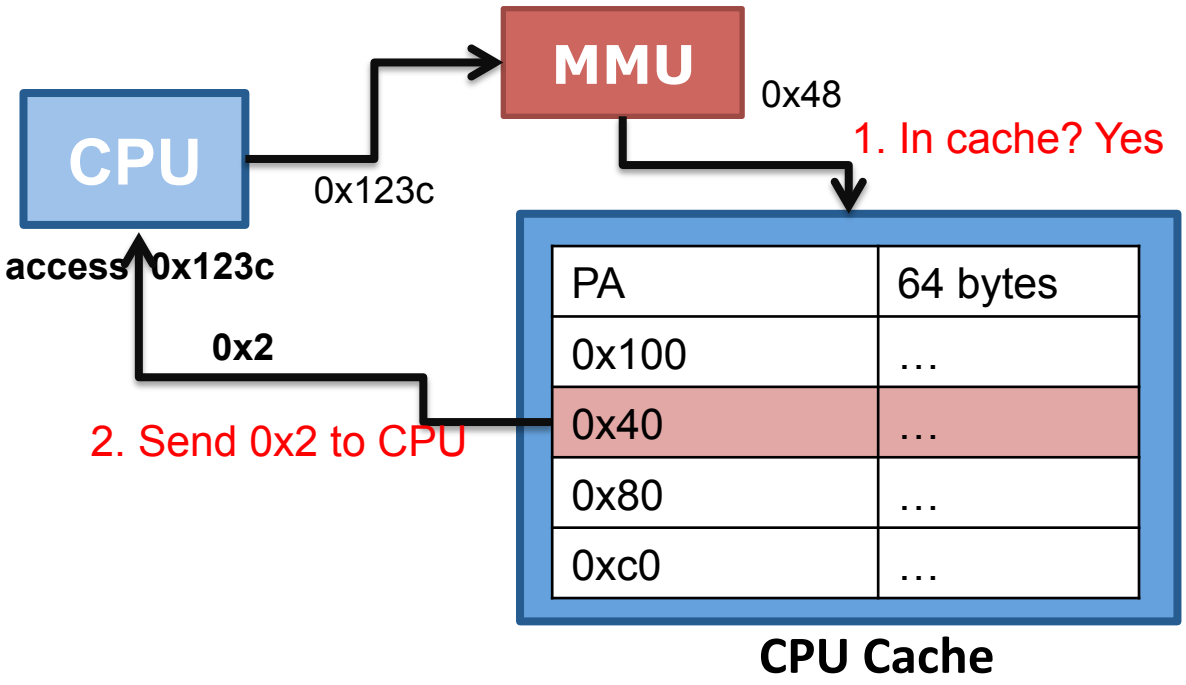
- Each cache line has 64 bytes

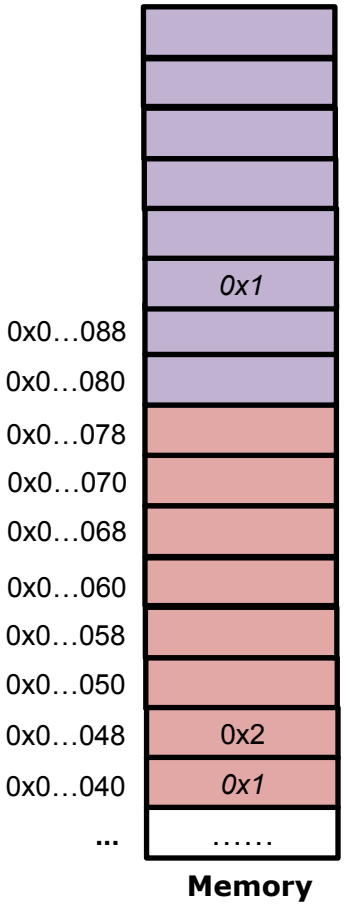


# Direct-mapped cache

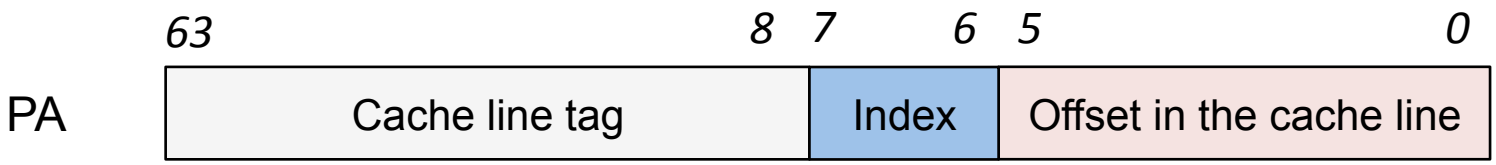
Caching at block granularity

- Each cache line has 64 bytes

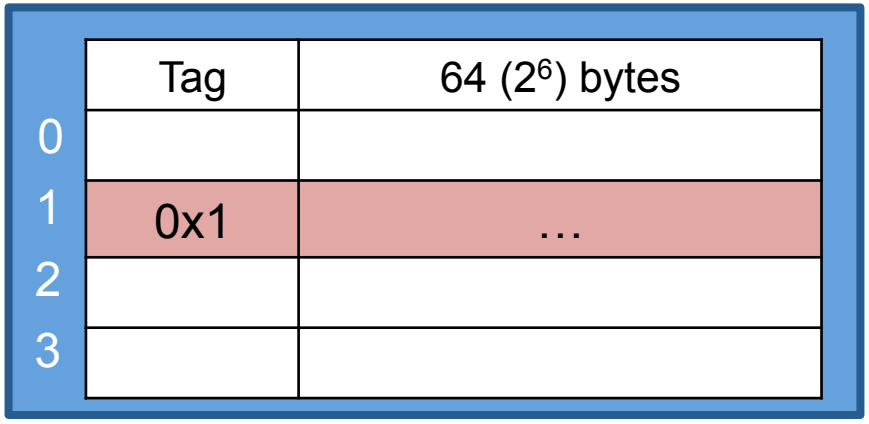


# Check and identify the location



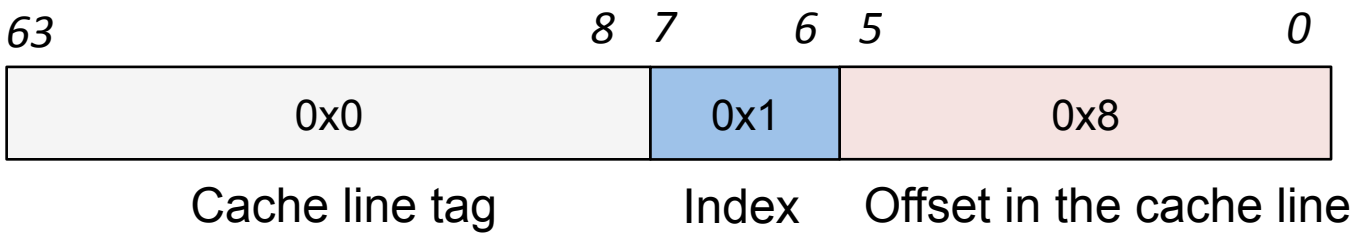

CPU access data at (PA)



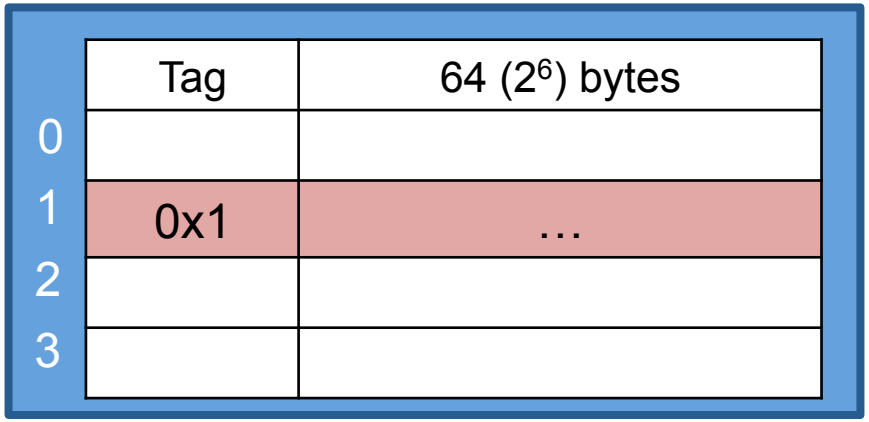

**CPU Cache (64 bytes cache line)**

1. Use the index bits[6:7] to find the the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63]
3. On cache hit, use the offset bits[0:5] to find the data in the cache line.
4. On cache miss, fetch the data from memory.

# Check and identify the location



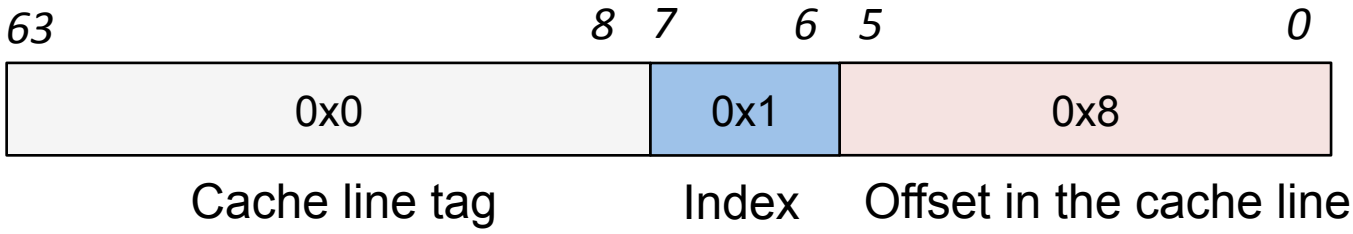

CPU access data at (0x48 PA)



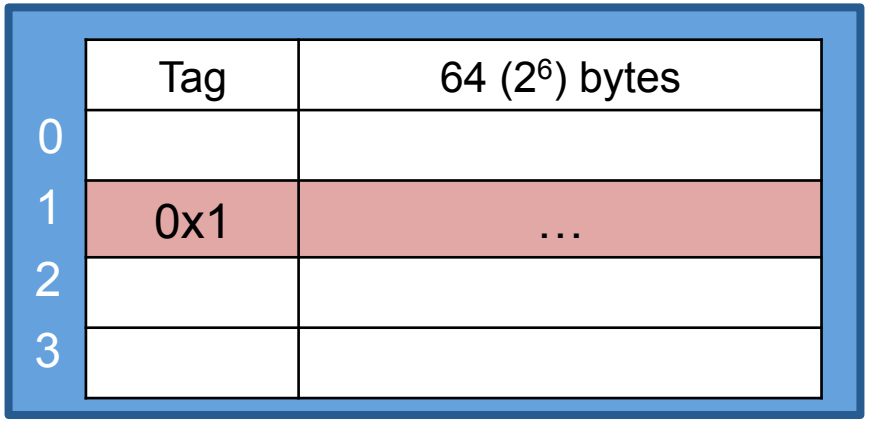

**CPU Cache (64 bytes cache line)**

1. Use the index bits[6:7] to find the the cache line which may buffer the data.

# Check and identify the location



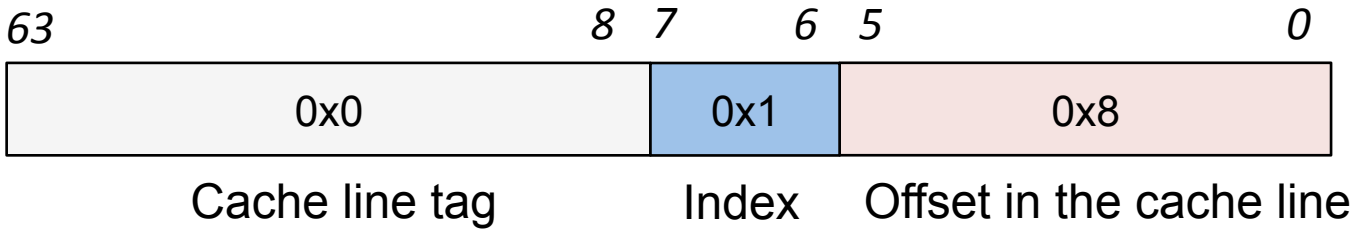

CPU access data at (0x48 PA)



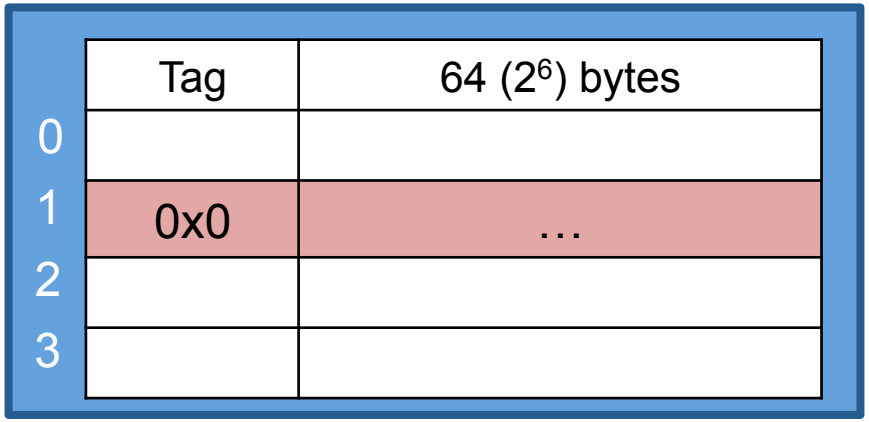

**CPU Cache (64 bytes cache line)**

1. Use the index bits[6:7] to find the the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63] (Cache miss)

# Check and identify the location



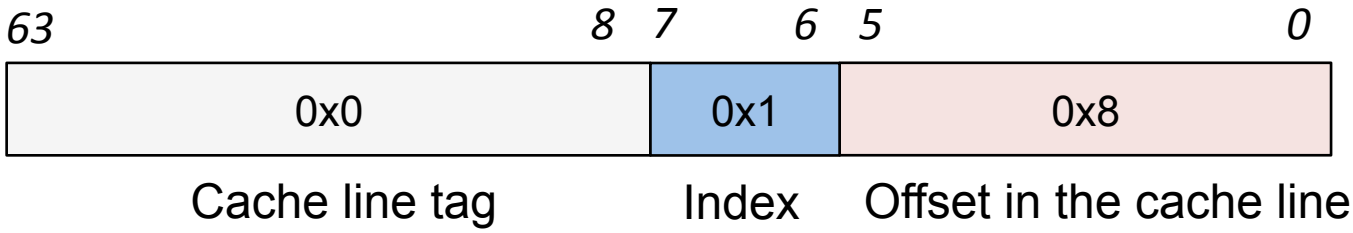

CPU access data at (0x48 PA)



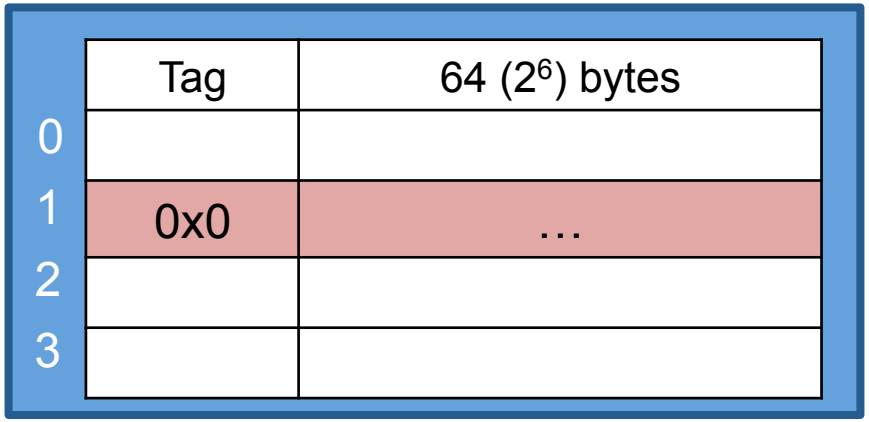

**CPU Cache (64 bytes cache line)**

1. Use the index bits[6:7] to find the the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63] (Cache miss)
3. Load 64 bytes from 0x40

# Check and identify the location




CPU access data at (0x48 PA)




**CPU Cache (64 bytes cache line)**

1. Use the index bits[6:7] to find the the cache line which may buffer the data.
2. Compare the cache line tag bits[8:63] (Cache miss)
3. Load 64 bytes from 0x40
4. Send the data at the offset of 0x8 in buffered cache line.

# Issue I

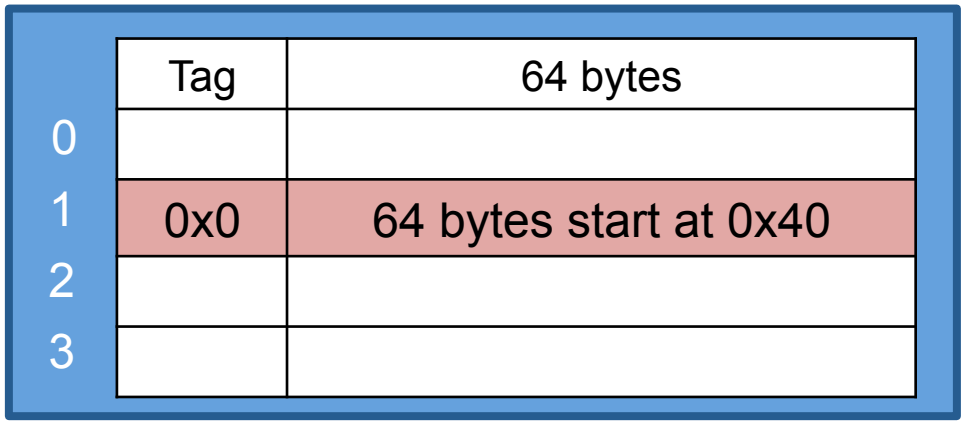
Access pattern:

→ access **0x40** **cache miss**

access **0x140**

access **0x40**

access **0x140**




**CPU Cache**

Cache line tag: 0

Cache line index: 1

Load 64 bytes start at 0x40 into 1st entry

# Issue I

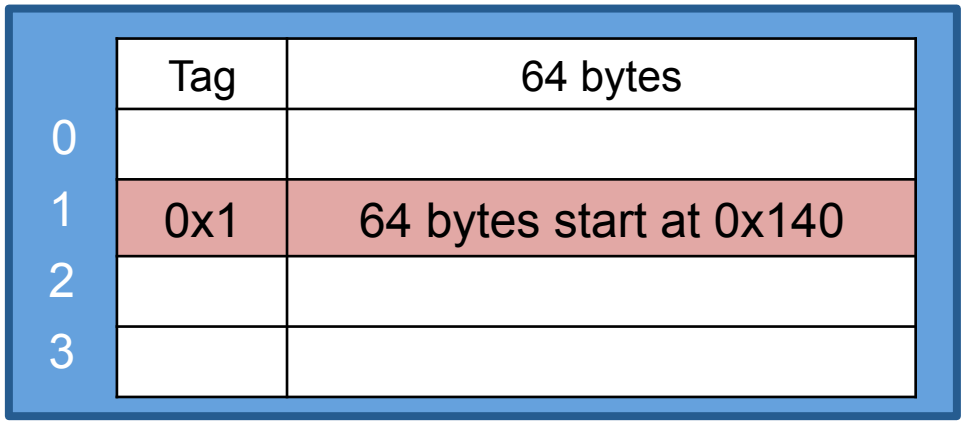
Access pattern:

access **0x40** **cache miss**

→ access **0x140** **cache miss**

access **0x40**

access **0x140**




**CPU Cache**

Cache line tag: 1

Cache line index: 1

Evict old cache line (1st entry),

Load 64 bytes start at 0x140 into 1st entry

# Issue I

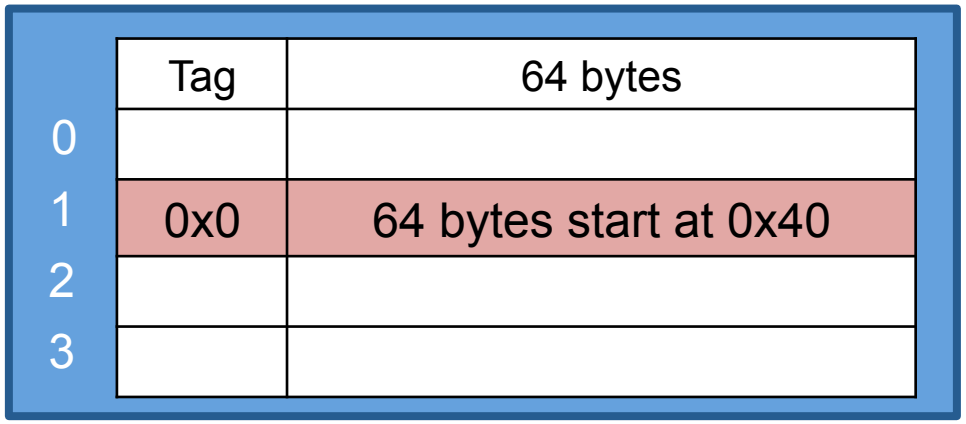
Access pattern:

access **0x40** **cache miss**

access **0x140** **cache miss**

→ access **0x40** **cache miss**

access **0x140**




**CPU Cache**

Cache line tag: 0

Cache line index: 1

Evict old cache line (1st entry),

Load 64 bytes start at 0x40 into 1st entry

# Issue I

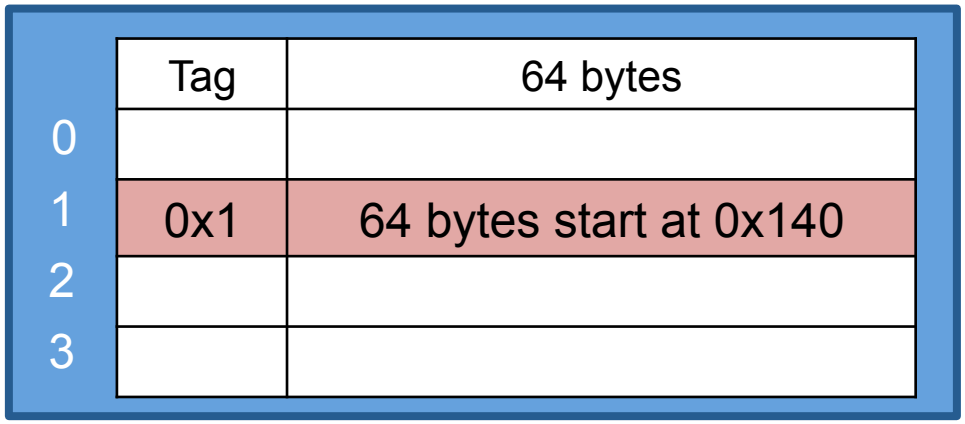
Access pattern:

access **0x40** **cache miss**

access **0x140** **cache miss**

access **0x40** **cache miss**

→ access **0x140** **cache miss**




**CPU Cache**

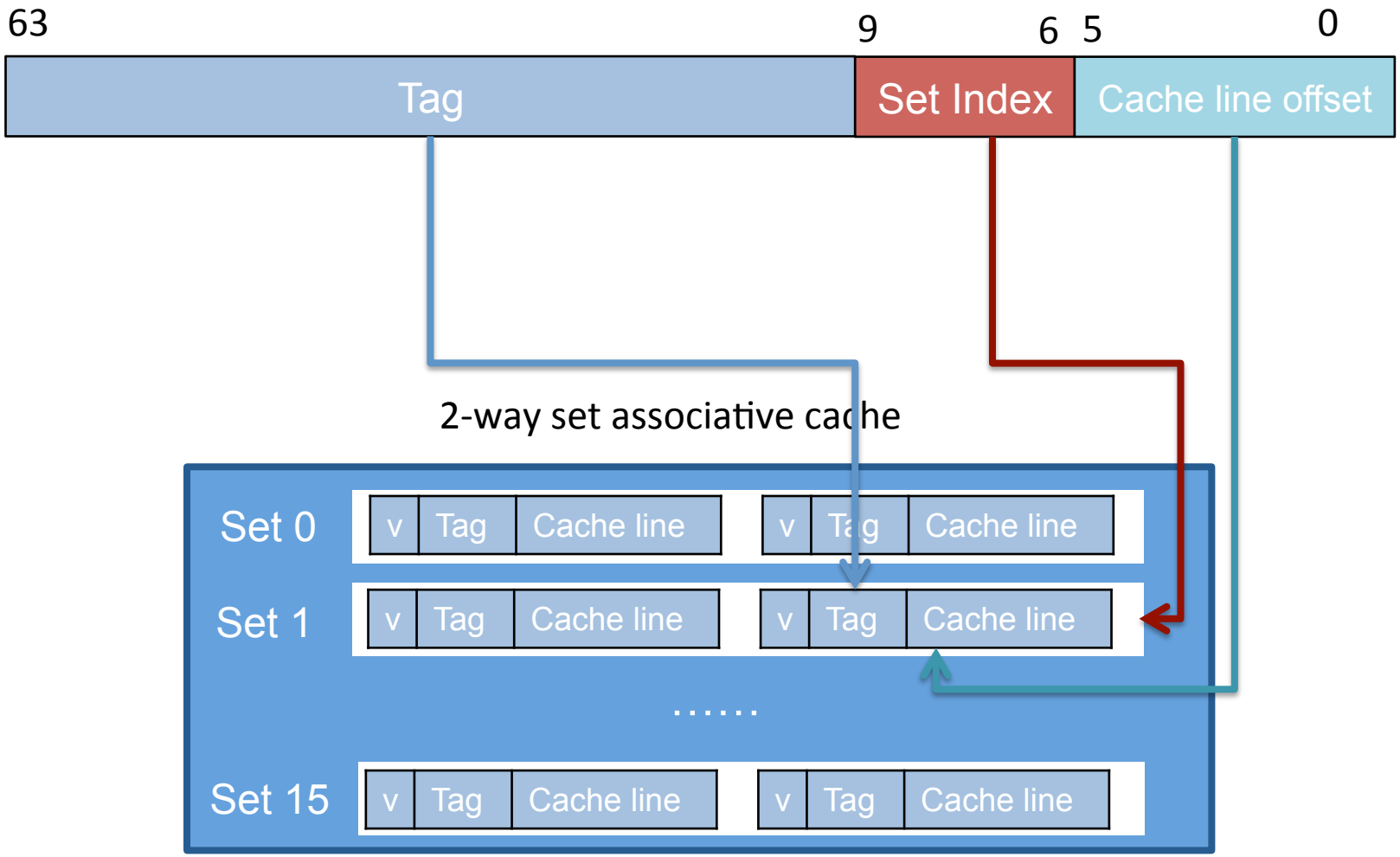
Cache line tag: 1

Cache line index: 1

Evict old cache line (1st entry),

Load 64 bytes start at 0x140 into 1st entry

# Multi-way set associative cache

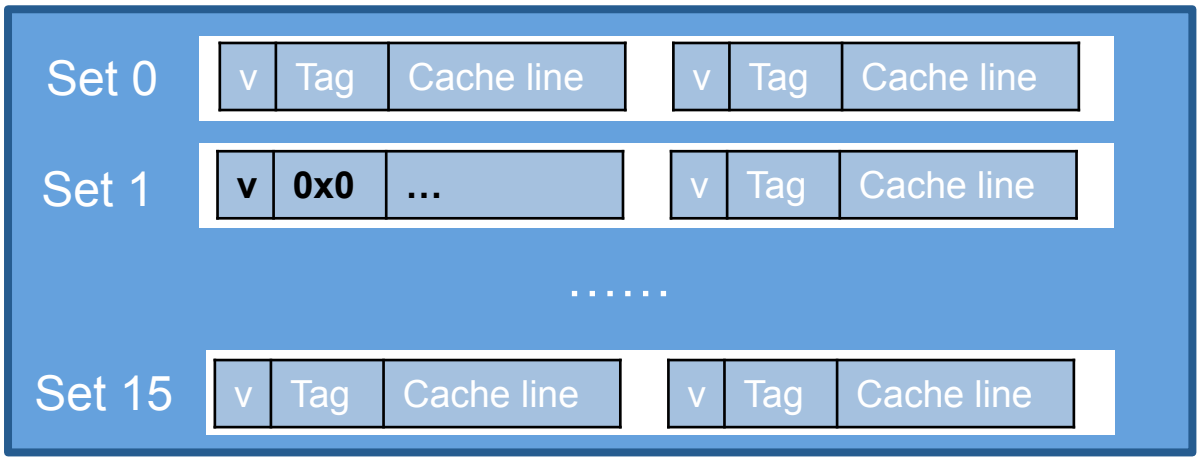



CPU Cache

# Multi-way set associative cache

Access pattern:

→ access **0x40** **miss**
access **0x440**
access **0x40**
access **0x440**


**CPU Cache**

# Multi-way set associative cache

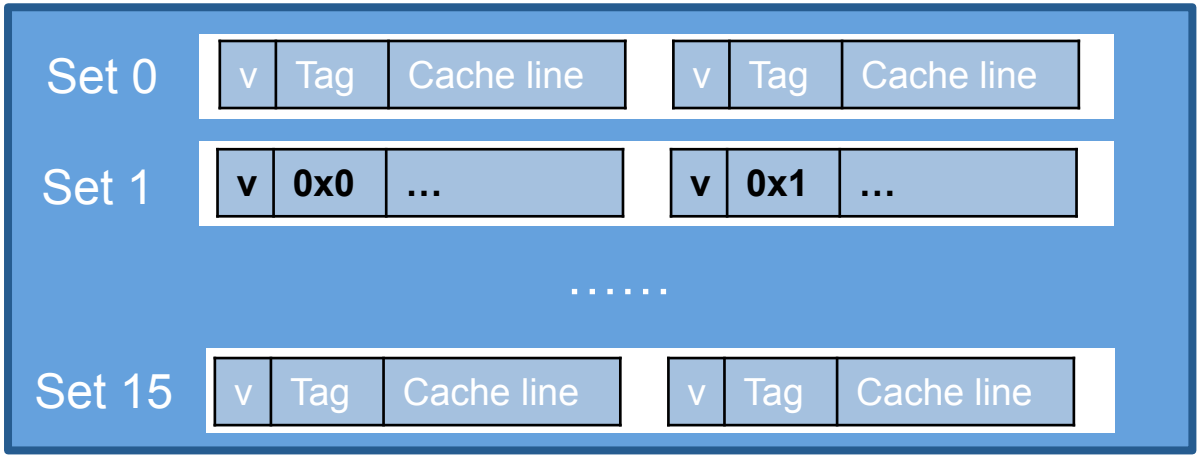
Access pattern:

access **0x40** **miss**

→ access **0x440** **miss**

access **0x40**

access **0x440**

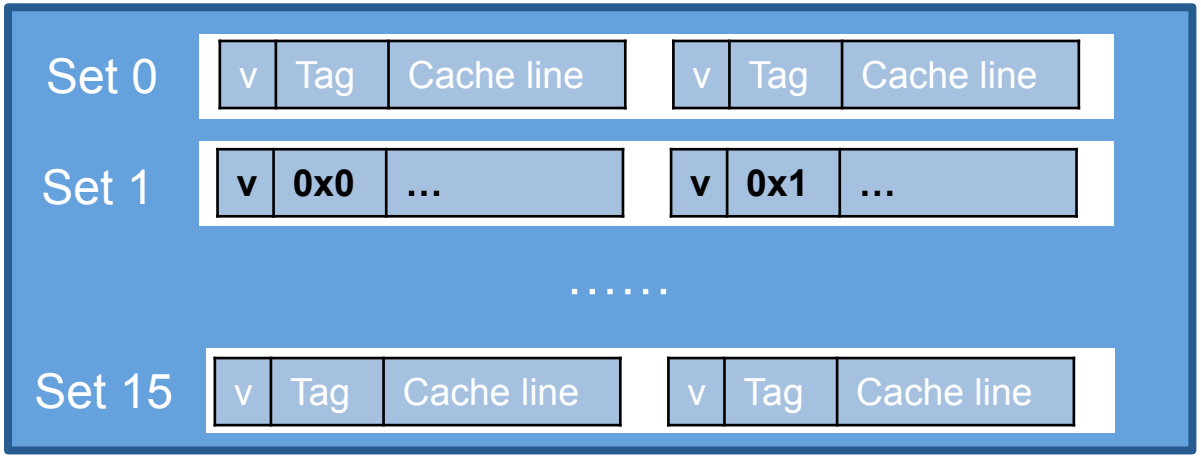




**CPU Cache**

# Multi-way set associative cache

Access pattern:

access **0x40** **miss**
access **0x440** **miss**
access **0x40** **hit**
access **0x440** **hit**

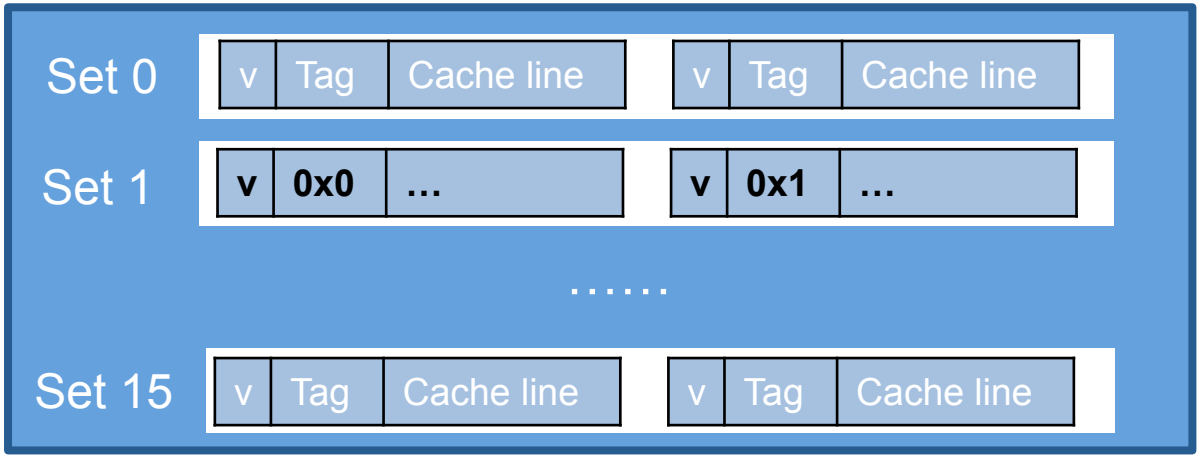

**CPU Cache**

# Multi-way set associative cache

Access pattern:

access **0x40**
access **0x440**
→ access **0x840**

**Which cache line in set 1 should be evicted?**


**CPU Cache**

# Cache line replacement policy

LFU (least-frequently-used)

- Replace the line that has been referenced the fewest times over some past time window

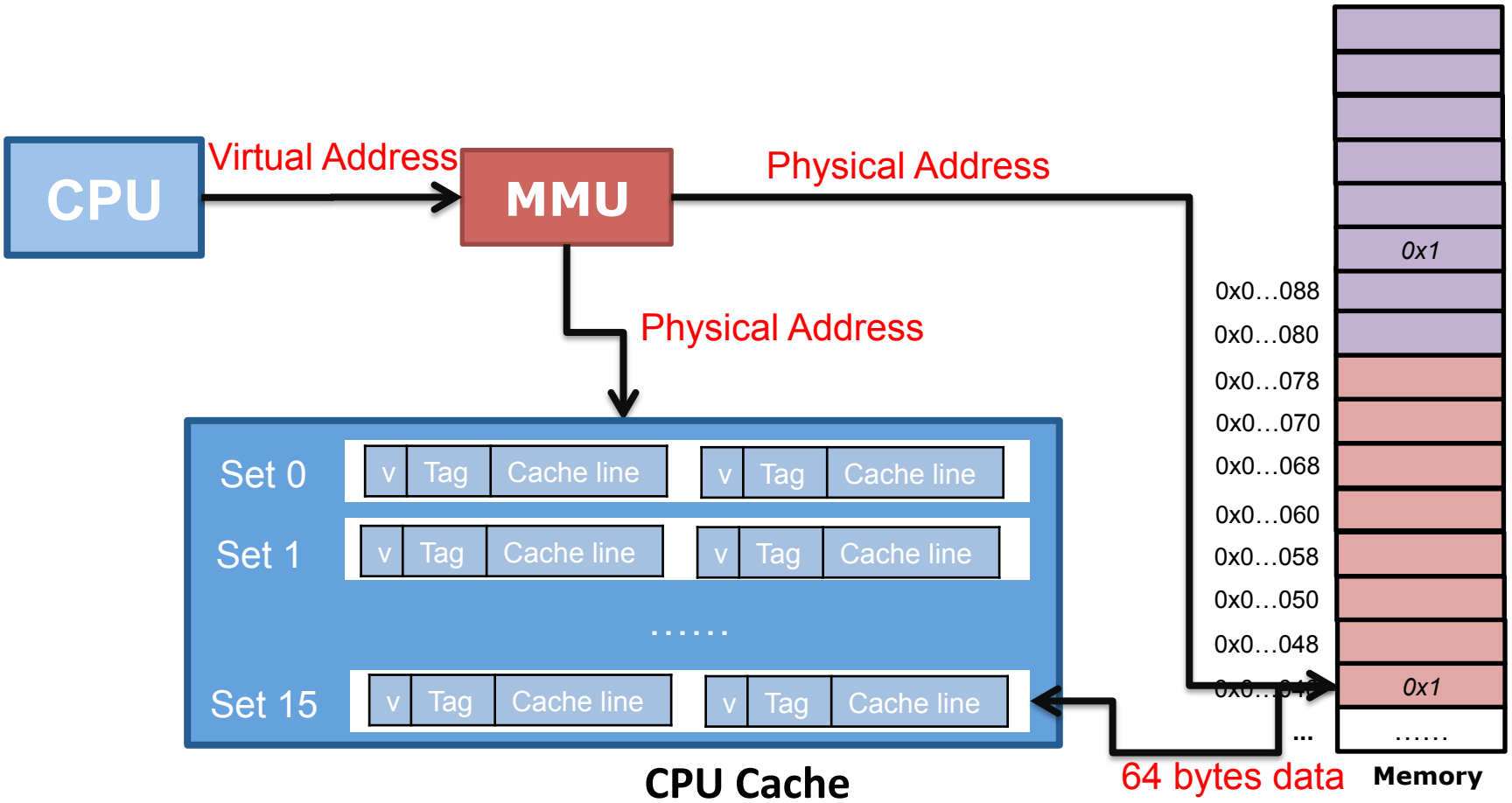
LRU (least-recently-used)

- Replace the line that has the furthest access in the past

All of these policies require additional time and hardware

# Issue II

Can not access cache before the address translation.

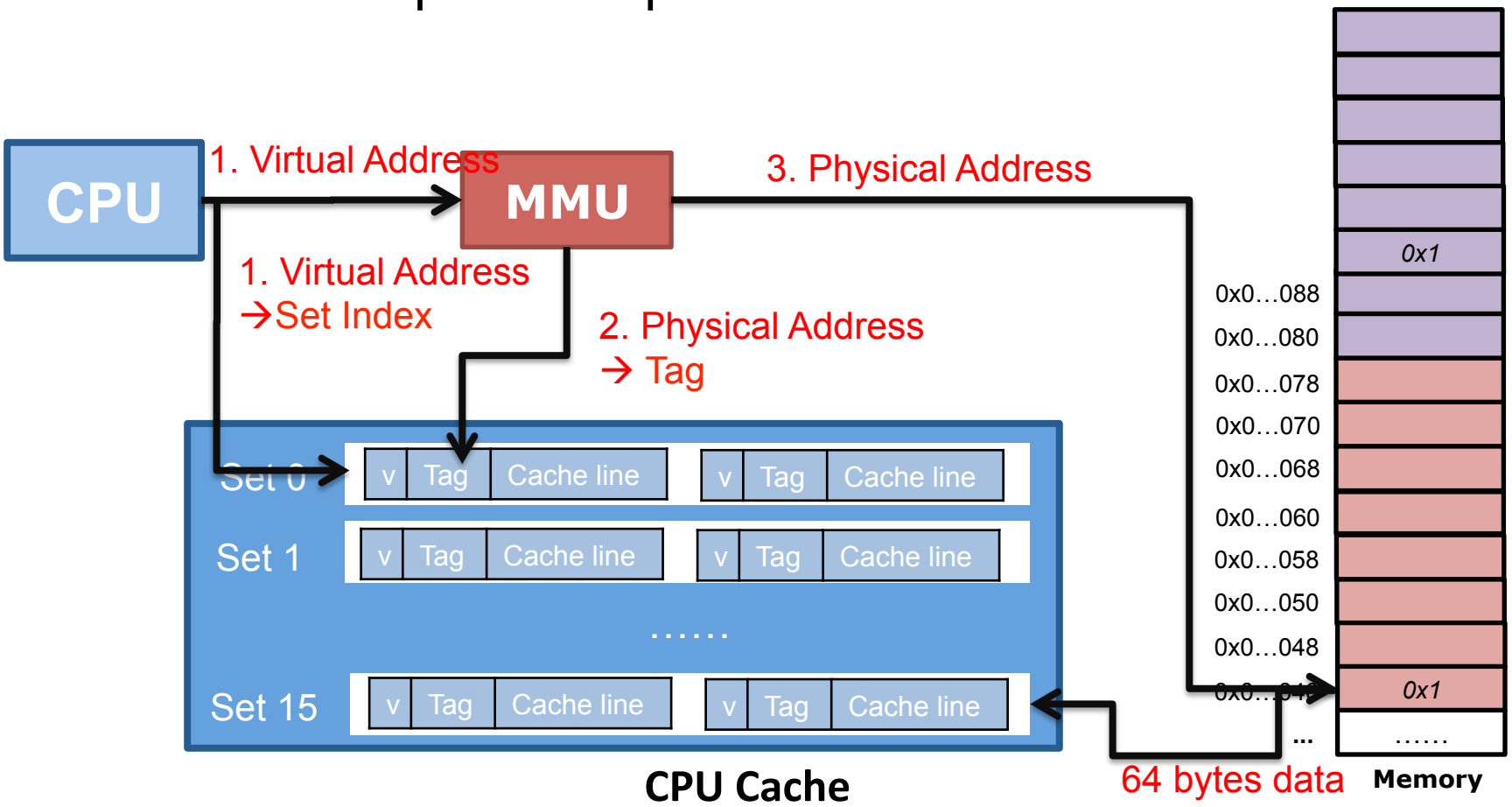


# Low latency cache

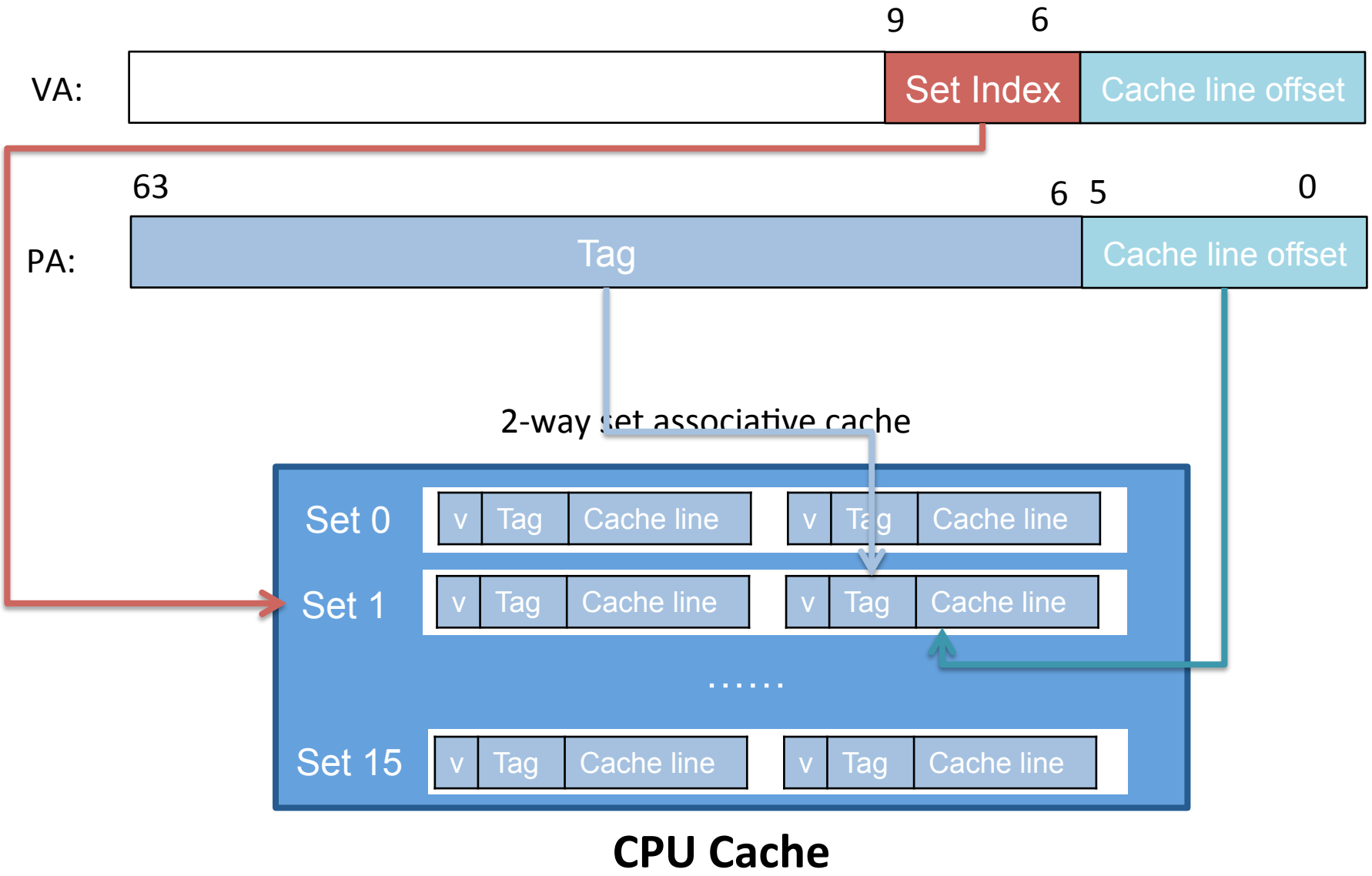
Virtual Index and Physical Tag

- Use VA to index set, calculate the tag from PA
- Cache set lookup can be parallel with address translation

2.

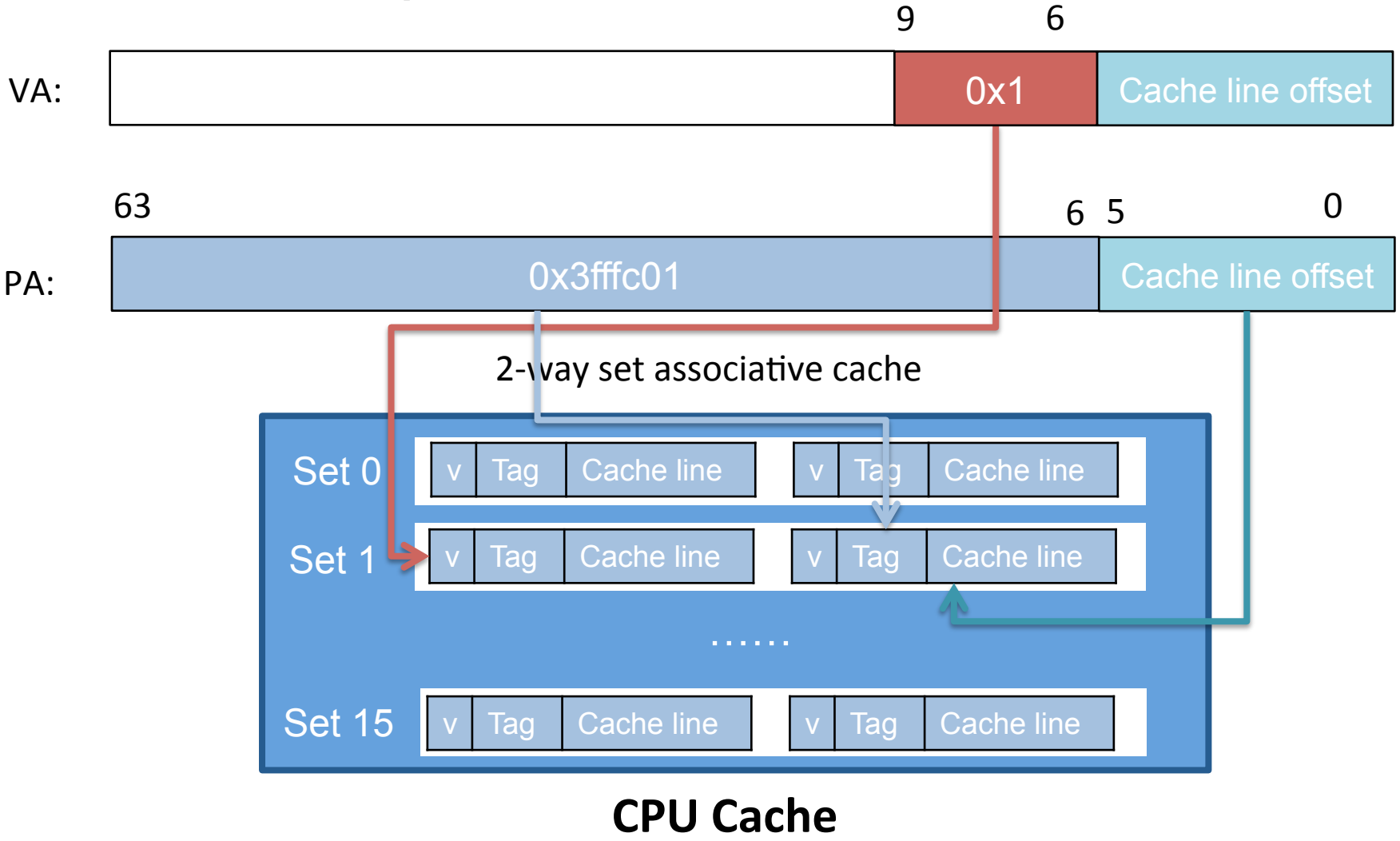


# Virtual Index and Physical Tag




**CPU Cache**

# Virtual Index and Physical Tag



# Memory hierarchy