Directmapped Cache, SumArrayRows

int a[2][8]

0x100 **A**	0x104 **B**	0x108 **C**	0x10C **D**	0x110 **E**	0x114 **F**	0x118 **G**	0x11C **H**
0x120 **I**	0x124 **J**	0x128 **K**	0x12C **L**	0x120 **M**	0x124 **N**	0x128 **O**	0x12C **P**

0x100
0x101
0x102 A
0x103

0b0001_0000_0000

0001_000

0x104
0x105
0x106 B
0x107

0b0001_0000_0100

0001_000

0x108 C

0b0001_0000_1000

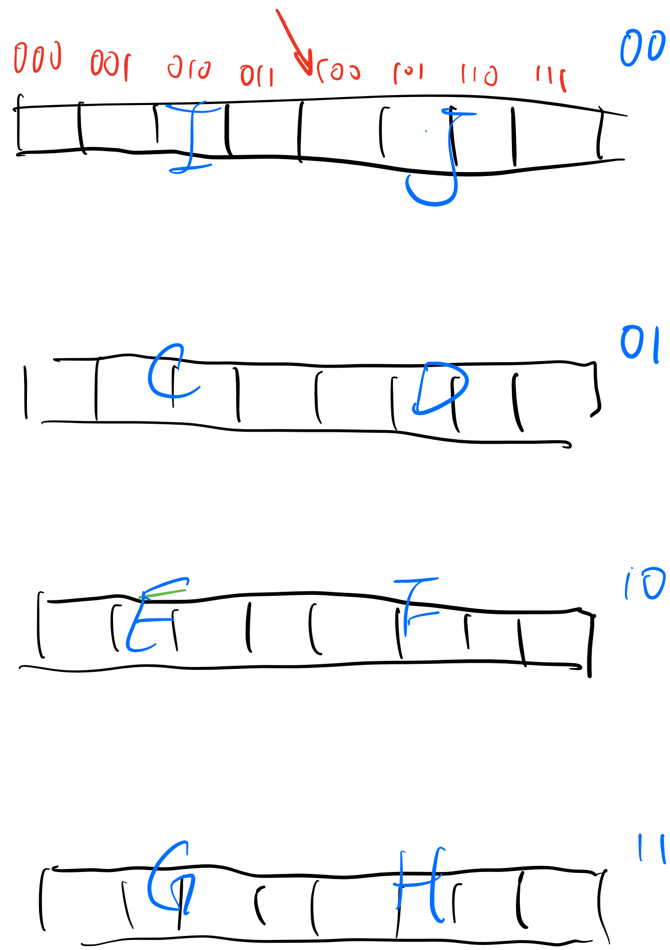
0x10C D

0b0001_0000_1100

0x120 I

0b0001_0010_0000

9



Directmapped Cache, sumArrayCols

int a[2][8]

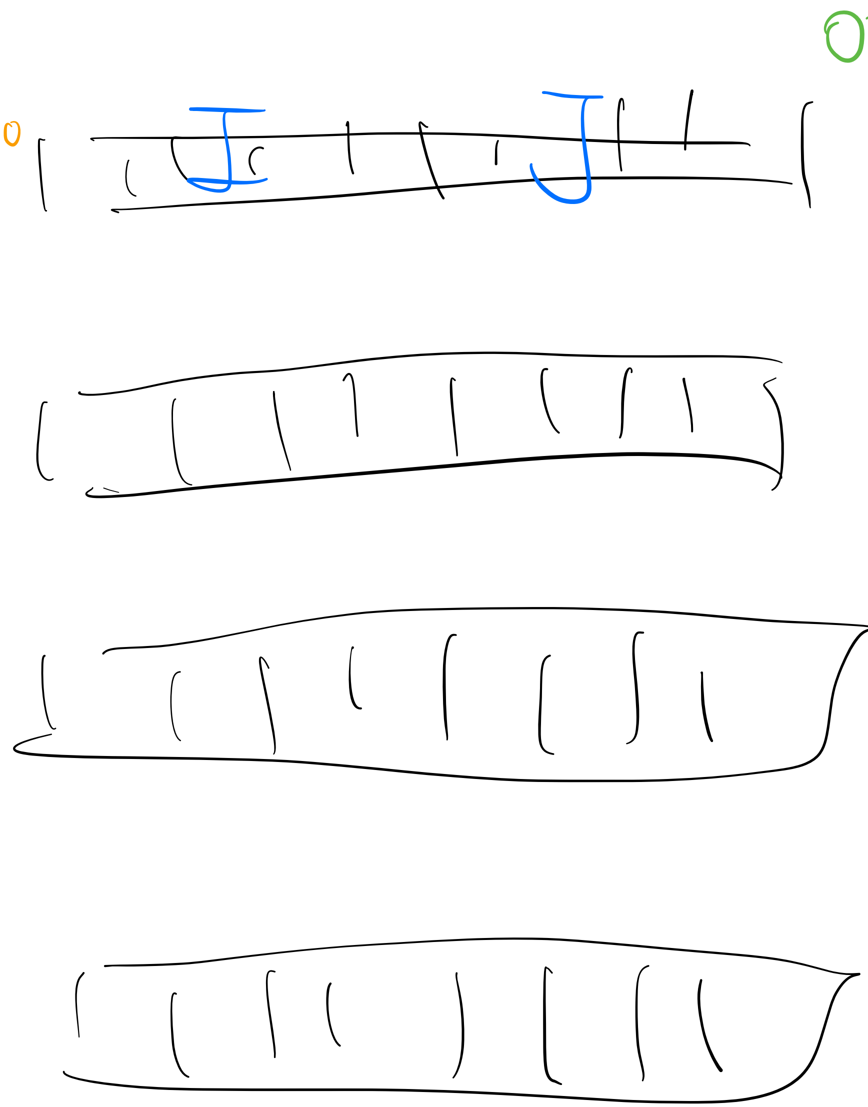
0x100 **A**	0x104 **B**	0x108 **C**	0x10C **D**	0x110 **E**	0x114 **F**	0x118 **G**	0x11C **H**
0x120 **I**	0x124 **J**	0x128 **K**	0x12C **L**	0x120 **M**	0x124 **N**	0x128 **O**	0x12C **P**

0x120 I

0001_000

0 0001_0010_0000

?



A m
I m, e
B m, e
J m, e
C m
K m, e
D m, e
L m, e
E
m

F
N
G
O
H
P

		fully associative cache	set associative cache	direct mapped cache
	s, the number of set index bits	0	1	2
	S=2^s, the number of sets	1	2	4
	E-way, the number of interchangable slots in each set	4	2	1
	b, the number of bits in the block offset	3	3	3
	B=2^b, the number of bytes in each block	8	8	8
	total capacity in bytes = S * E * B	32 bytes	32 bytes	32 bytes
sumArrayRows	total hits	8	8	8
sumArrayRows	total misses	8	8	8
sumArrayRows	total eviction	4	4	4
sumArrayCols	total hits	8	8	0
sumArrayCols	total misses	8	8	16
sumArrayCols	total eviction	4	4	12
	analogous data structure	for FIFO, like a queue	a hash table of queues (assuming FIFO)	hash table using set as hash
	positive attributes	high associativity offers better support for temporal locality		least cost in search time hardware area power coonsumption
	negative attributes	comparing tags for all ways has high costs in time, silicon area, power consumption		strict direct mapping of memory addresses to specific sets leads to conflict misses

The memory hierarchy: Cache placement, replacement, and write policies

Yipeng Huang

Rutgers University

November 18, 2025

Table of contents

PA5: Simulating a cache and optimizing programs for caches

Write a cache simulator

1. fullyAssociative
2. directMapped
3. setAssociative

Optimize some code for better cache performance

1. cacheBlocking
2. cacheOblivious

PA5: Simulating a cache and optimizing programs for caches

A tour of files in the package

- trace files
- csim-ref

Table of contents

Locality: How to create illusion of fast access to capacious data

From the perspective of memory hierarchy, locality is using the data in at any particular level more frequently than accessing storage at next slower level.

First, let's experience the puzzling effect of locality in `sumArray.c`

- sumArrayRows()
- sumArrayCols()

Well-written programs maximize locality

- Spatial locality
- Temporal locality

Spatial locality

```
double dotProduct (
  double a[N],
  double b[N],
) {
  double sum = 0.0;
  for(size_t i=0; i<N; i++){
    sum += a[i] * b[i];
  }
  return sum;
}
```

Spatial locality

- Programs tend to access adjacent data.
- Example: stride 1 memory access in `a` and `b`.

Temporal locality

```c
double dotProduct (
  double a[N],
  double b[N],
) {
  double sum = 0.0;
  for(size_t i=0; i<N; i++){
    sum += a[i] * b[i];
  }
  return sum;
}
```

Temporal locality

- Programs tend to access data over and over.
- Example: `sum` gets accessed N times in iteration.

Table of contents

Cache placement policy (how to find data at address for read and write hit)

Several designs for caches

- Fully associative cache
- Direct-mapped cache
- N-way set-associative cache

Cache design options use m-bit memory addresses differently

- t-bit tag
- s-bit set index
- b-bit block offset

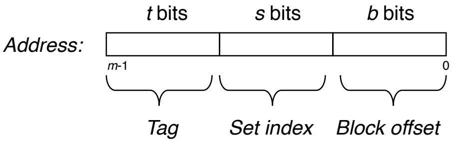


Figure: Memory addresses. Image credit CS:APP

Fully associative cache

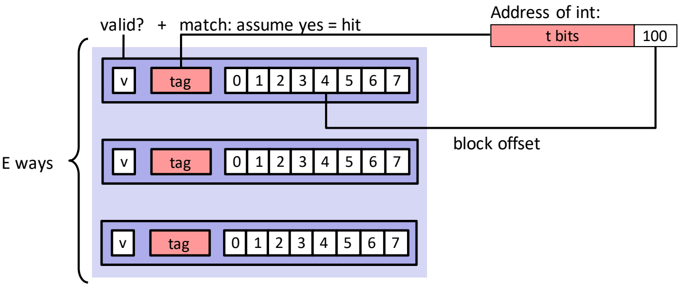


Figure: Fully associative cache. Image credit CS:APP

m-bit memory address split into:

- *t*-bit tag
- *b*-bit block offset

Fully associative cache

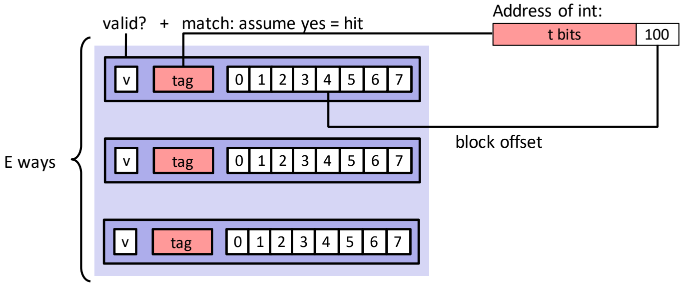


Figure: Fully associative cache. Image credit CS:APP

b-bit block offset

- here, $b = 3$
- The number of bytes in a block is $B = 2^b = 2^3 = 8$
- A block is the minimum number of bytes that can be cached
- Good for capturing spatial locality, short strides

Fully associative cache

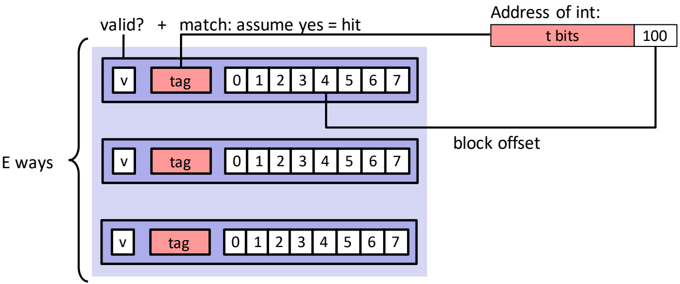


Figure: Fully associative cache. Image credit CS:APP

t-bit tag

- here,
  $t = m - b = m - 3$
- When CPU wants to read from or write to memory, all t-bits in tag need to match for read/write hit.

Fully associative cache

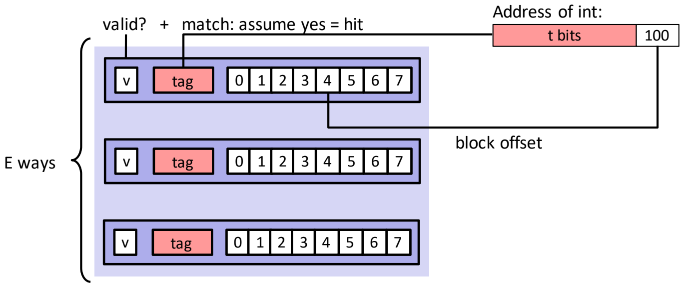


Figure: Fully associative cache. Image credit CS:APP

Full associativity

- Blocks can go into any of E ways
- Here, $E = 3$
- Good for capturing temporal locality: cache hits can happen even with intervening reads and writes to other tags.

Fully associative cache

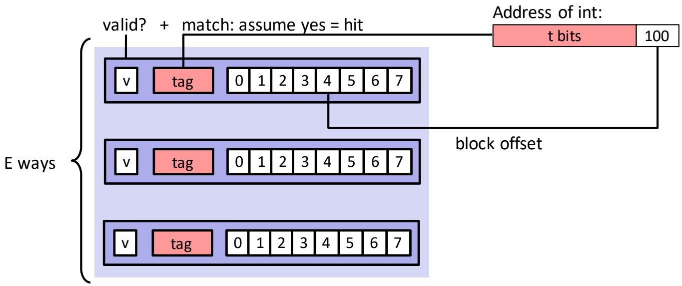


Figure: Fully associative cache. Image credit CS:APP

Capacity of cache

- Total capacity of fully associative cache in bytes: $C = EB = E * 2^b$
- Here, $C = E * 2^b = 3 * 2^3 = 24$ bytes

Fully associative cache

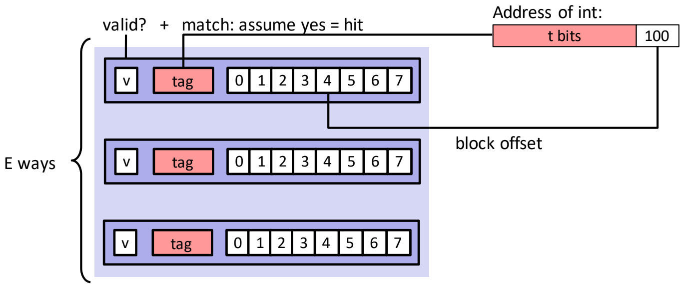


Figure: Fully associative cache. Image credit CS:APP

Strengths

- Blocks can go into any of *E*-ways.
- Hit rate is only limited by total capacity.

Weaknesses

- Searching across all valid tags is expensive.
- Figuring out which block to evict on read/write miss is also expensive.
- Requires a lot of

Direct-mapped cache

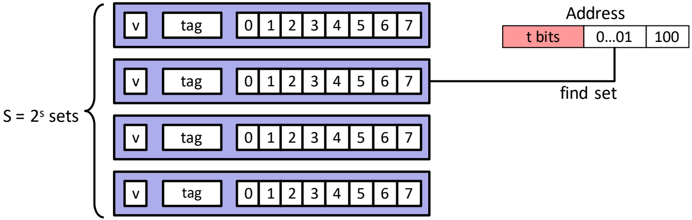


Figure: Direct-mapped cache. Image credit CS:APP

m-bit memory address split into:

- *t*-bit tag
- *s*-bit set index
- *b*-bit block offset

Direct-mapped cache

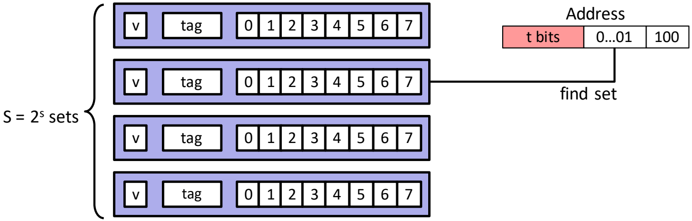


Figure: Direct-mapped cache. Image credit CS:APP

b-bit block offset

- here, $b = 3$
- The number of bytes in a block is $B = 2^b = 2^3 = 8$
- A block is the minimum number of bytes that can be cached
- Good for capturing spatial locality, short strides

Direct-mapped cache

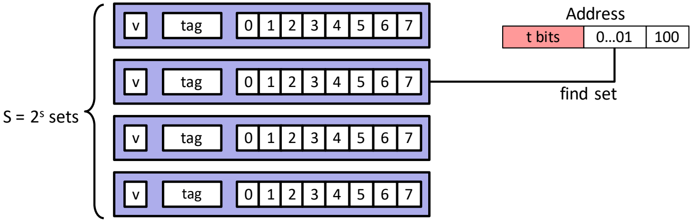


Figure: Direct-mapped cache. Image credit CS:APP

s-bit set index

- here, $s = 2$
- The number of sets in cache is $S = 2^s = 2^2 = 4$
- A hash function that limits exactly where a block can go
- Good for further increasing ability to exploit spatial locality, short strides

Direct-mapped cache

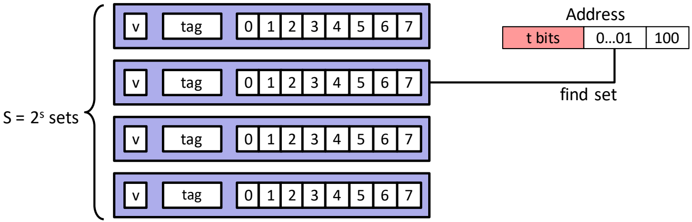


Figure: Direct-mapped cache. Image credit CS:APP

t-bit tag

- here,
  $t = m - s - b = m - 2 - 3$
- When CPU wants to read from or write to memory, all t-bits in tag need to match for read/write hit.

Direct-mapped cache

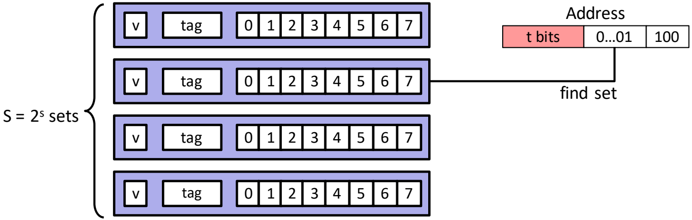


Figure: Direct-mapped cache. Image credit CS:APP

Direct mapping

- In direct-mapped cache, blocks can go into only one of $E = 1$ way

Direct-mapped cache

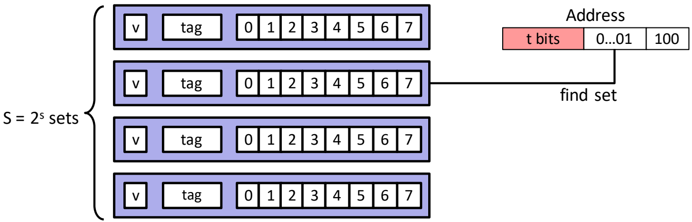


Figure: Direct-mapped cache. Image credit CS:APP

Capacity of cache

- Total capacity of fully associative cache in bytes: $C = SEB = 2^s * E * 2^b$
- Here, $C = 2^s * E * 2^b = 2^2 * 1 * 2^3 = 32$ bytes

Direct-mapped cache

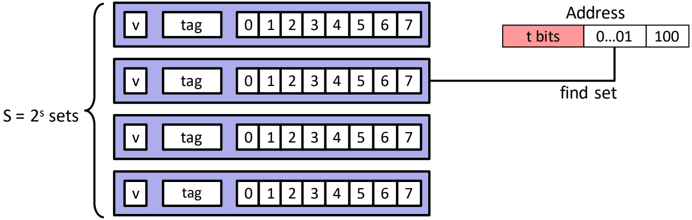


Figure: Direct-mapped cache. Image credit CS:APP

Strengths

- Simple to implement.
- No need to search across tags.

Weaknesses

- Can lead to surprising thrashing of cache with unfortunate access patterns.
- Unexpected conflict misses independent of cache capacity.

E-way set-associative cache

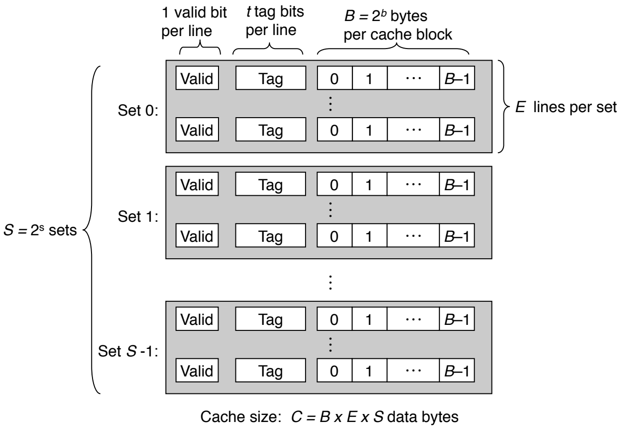


Figure: Direct-mapped cache. Image credit CS:APP

Strengths

- Blocks can go into any of E-ways, increases ability to support temporal locality, thereby increasing hit rate.
- Only need to search across E tags. Avoids costly searching across all valid tags.
- Avoids conflict misses due to unfortunate access patterns.

E-way set-associative cache

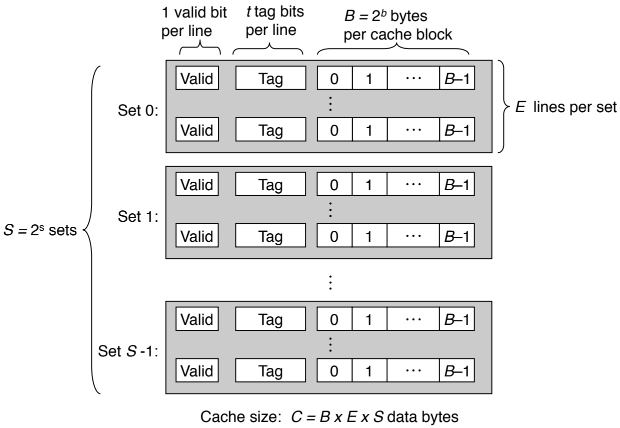
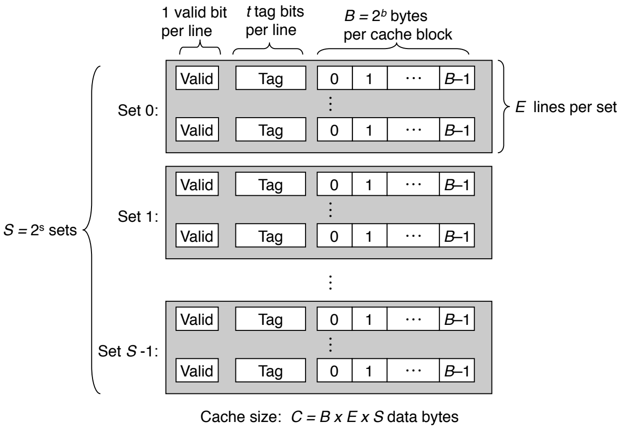


Figure: Direct-mapped cache. Image credit CS:APP

Used in practice in, e.g., a recent Intel Core i7:

- $C = 32$KB L1 data cache per core
- $S = 64 = 2^6$ sets/cache ($s = 6$ bits)
- $E = 8 = 2^3$ ways/set
- $B = 64 = 2^6$ bytes/block ($b = 6$ bits)
- $C = S * E * B$
- Assuming memory addresses are $m = 48$, then tag size $t = m - s - b = 48 - 6 - 6 = 36$ bits.

E-way set-associative cache



Let's see textbook slides for a simulation

Figure: Direct-mapped cache. Image credit CS:APP

Table of contents

Cache hits

Memory access is serviced from cache

- Hit rate = $\frac{Number of hits}{Number of memory accesses}$
- Hit time: latency to access cache (4 cycles for L1, 10 cycles for L2)

Cache misses: metrics

Memory access cannot be serviced from cache

- Miss rate = $\frac{Number of misses}{Number of memory accesses}$
- Miss penalty (miss time): latency to access next level cache or memory (up to 200 cycles for memory).
- Average memory access time = hit time + miss rate $\times$ miss penalty

Cache misses: Classification

Compulsory misses

- First access to a block of memory will miss because cache is cold.

Conflict misses

- Multiple blocks map (hash) to the same cache set.
- Fully associative caches have no such conflict misses.

Capacity misses

- Occurs when the set of active cache blocks (working set) is larger than the cache.
- Direct mapped caches have no such capacity misses.

Table of contents

Direct-mapped cache

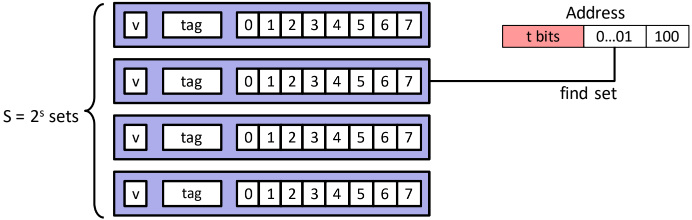


Figure: Direct-mapped cache. Image credit CS:APP

No need for replacement policy

- The number of sets in cache is $S = 2^s = 2^2 = 4$.
- A hash function that limits exactly where a block can go.
- In direct-mapped cache, blocks can go into only one of $E = 1$ way.
- No cache replacement policy is needed.

Associative caches

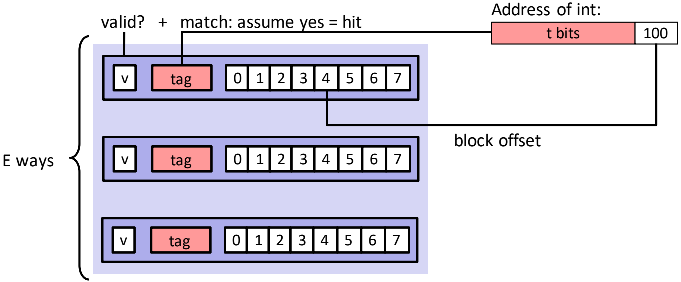


Figure: Fully associative cache. Image credit CS:APP

Needs replacement policy

- Blocks can go into any of E ways
- Here, $E = 3$
- Good for capturing temporal locality.
- If all ways/lines/blocks are occupied, and a cache miss happens, which way/line/block will be the victim and get evicted for replacement?

Cache replacement policies for associative caches

FIFO: First-in, first-out

- Evict the cache line that was placed the longest ago.
- Each cache set essentially becomes limited-capcity queue.

LRU: Least Recently Used

- Evict the cache line that was last accessed longest ago.
- Needs a counter on each cache line, and/or a global counter (e.g., program counter).

Table of contents

Policies for writes from CPU to memory

How to deal with write-hit?

- **Write-through.** Simple. Writes update both cache and memory. Costly memory bus traffic.
- **Write-back.** Complex. Writes update only cache and set a dirty bit; memory updated only upon eviction. Reduces memory bus traffic. (For multi-core CPUs, motivates complex cache coherence protocols)

How to deal with write-miss?

- **No-write-allocate.** Simple. Write-misses do not load block into cache. But write-misses are not mitigated via cache support.
- **Write-allocate.** Complex. Write-misses will load block into cache.

Typical designs:

- **Simple:** write-through + no-write-allocate.
- **Complex:** write-back + write-allocate.

Table of contents

Multilevel cache hierarchies

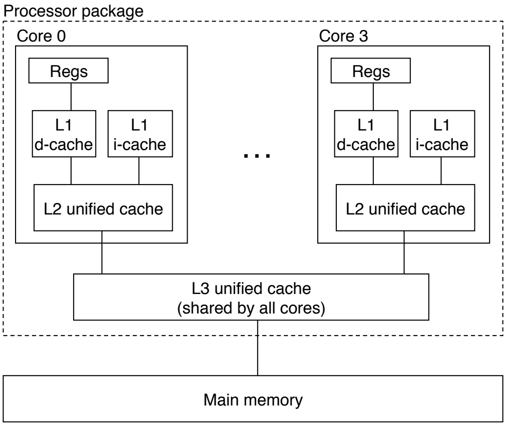


Figure: Intel Core i7 cache hierarchy. Image credit CS:APP

Small fast caches nested inside large slow caches

- L1 data and instruction cache: 32KB, 64 set, 8-way associative, 64B block, 4 cycle latency.
- L2 cache: 256KB, 512 set, 8-way associative, 64B block, 10 cycle latency.
- L3 cache: 8MB, 8192 set, 16-way associative, 64B block, 40-75 cycle latency.

Notice how latency cost increases as *E*-way associativity increases.

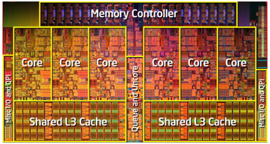


Figure: Intel 2020 Gulftown die shot. Image credit AnandTech