CPU Cache

Matthieu de Laharpe

Contents

1. General definitions
    1.1. Cache
    1.2. Cache hit & miss
2. CPU cache
    2.1. CPU cache hierarchy
    2.2. CPU cache level characteristics
    2.3. Cache entry structure
3. Cache mapping
    3.1. Direct-mapped
    3.2. Fully associative
    3.3. Set-associative
4. Cache policies
    4.1. Writing policies
    4.2. Replacement policies
5. Cache coherence
6. Going further

1.1- General definitions: cache

- Hardware or software component.
- Stores the result of expensive operations or computations:
  - I/O operations : opened file cached in RAM to limit hard drive operations,
  - Expensive function calls : lookup tables using parameters = memoization,
  - Memory access,
  - …
- Accelerate access to that data.

1.2- General definitions: cache hit & miss

- Cache hit: request for data that was stored in the cache.
- Cache miss: request for data that was not stored in the cache.
  - Compulsory miss: first ever request for the data, cache miss is inevitable !
  - Capacity miss: the cache is not large enough to hold the working set
  - Conflict miss: the requested data was previously evicted from the cache
  - Coherence miss: the requested data was invalidated by another thread
  - Coverage miss: the requested data was invalidated by the coherence directory

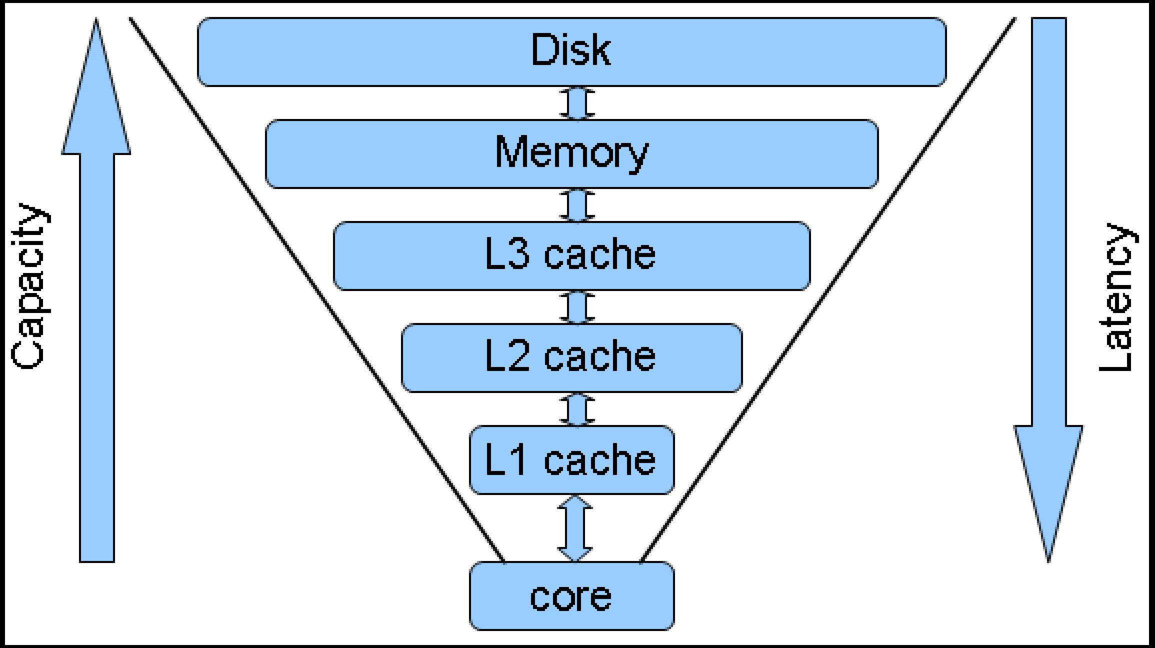
2- CPU cache

- Hardware component
- Used since the 1980s
- Lies between the main memory and the CPU registers
- Bridges operating frequency gap between CPU and main memory
- Uses static rather than dynamic RAM (SRAM/DRAM)
  - Expensive
  - Low power consumption
  - No refresh needed
- Benefits from locality of reference (both spatial and temporal)

2.1- CPU cache hierarchy

- Modern CPUs have multiple level of cache
  - Most commonly 2 or 3

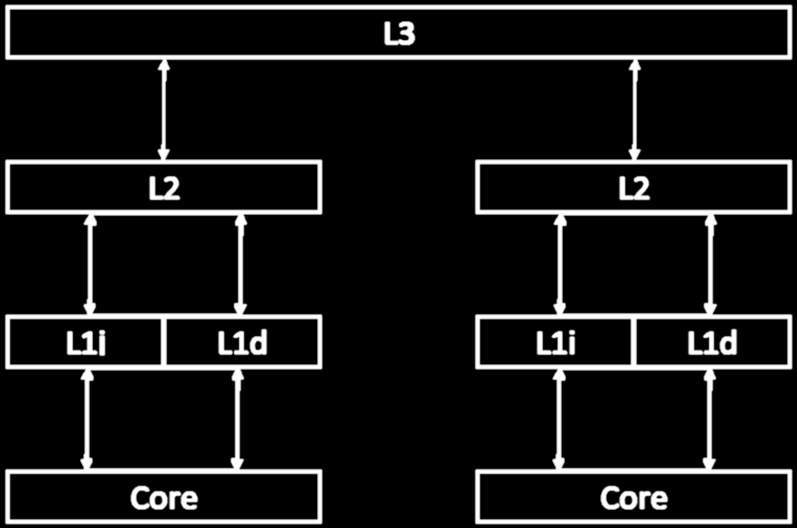
- Level 1 cache is split:
  - L1d -> data cache
  - L1i -> instruction cache



2.2- CPU cache level characteristics

Each cache level can be:

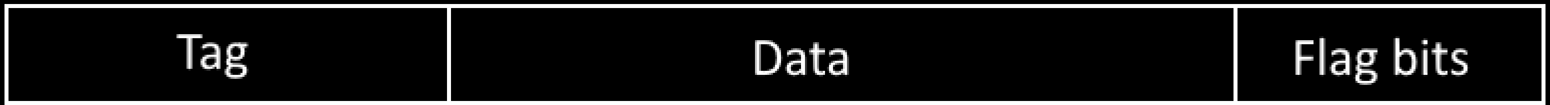
- Shared or Local
- Separate or Unified



- Inclusive,
  - Ln+1 contains data stored in Ln
  - Remove data by checking high level caches only
  - Echo cache line eviction to lower level caches
- Exclusive,
  - Ln+1 does NOT contain data stored in Ln
  - Stores more data
  - Both cache must have same line size
- or neither
  - Ln+1 may or may not contain data stored in Ln

2.3- Cache entry structure

- Data is transferred between the main memory and the cache in fixed size blocks called *cache lines* or *cache blocks*
  - Size between 32 and 256 bytes, most commonly 64 bytes
  - Size may vary between cache levels, higher size reduce compulsory miss rate (spatial locality)
- When a cache line is copied from the main memory to the cache, a cache entry is created. It holds:
  - The copied data,
  - A tag that identifies the entry,
  - Flag bits describing the status of the cache entry (validity, dirtiness …)



Tag	Data	Flag bits

- Only the size of the data counts toward the cache total size
  - A 256 KiB cache holds 256KiB of data !

3- Cache mapping

When we copy data from the main memory, where should it go ?

- 1 possible destination -> direct-mapped cache
  - Simplest design
  - More conflict misses and cache thrashing

- Every possible destination -> fully associative cache
  - Costly with large caches: must check EVERY cache block
  - No conflict misses, only capacity

- A compromise between the two -> set-associative cache
  - A main memory block has 2 to 16 possible destinations
  - Fewer conflict misses

3.1- Direct-mapped cache

Example:

- Main memory:
  - 256B (addresses: 0 to 255)
- Cache:
  - 32B direct-mapped
  - 8 cache lines
  - 4B line size
  - 3bits tag

	Tag	Data	Valid Bit
0	-	-	0
1	-	-	0
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.1- Direct-mapped cache: loading memory

- CPU request data from main memory with address 155.

155 ⇔ 10011011

- Block size is 4B = 2^2B, set 2 least significant bits of address to 0 to find block address

10011011 & 11111100 = 10011000 ⇔ 152

- Right shift block address by 2 to find block index in main memory (equ. to division by 4)

10011000 >> 2 = 00100110 ⇔ 38

Example:

- Main memory: 256B (addresses: 0 to 255)

address	151	152	153	154	155	156
data	5 B	7 1	0 F	8 4	E 3	9 2

memory block

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
1	-	-	0
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.1- Direct-mapped cache: loading memory

- There are $8 = 2^3$ cache lines, keep the 3 least significand bits of that index to find the cache entry index (equ. to mod 8)

00100110 & 00000111 = 00000110 ⇔ 6

- Use the remaining bits as the tag.

(00100110 & 11111000) >> 3 = 00000100 ⇔ 4

Example:

- Main memory: 256B (addresses: 0 to 255)

address	151	152	153	154	155	156
data	5 B	7 1	0 F	8 4	E 3	9 2

memory block

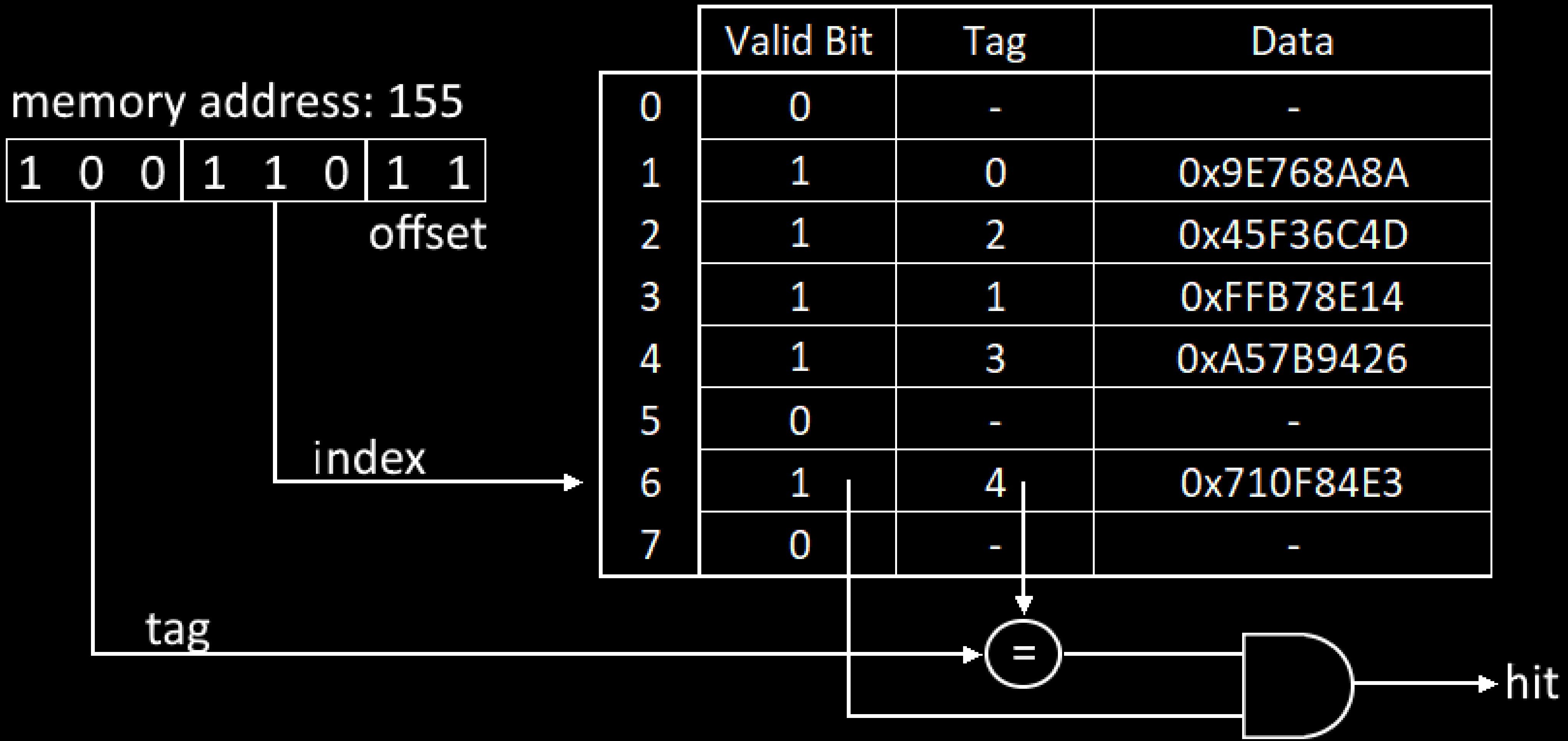
- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
1	-	-	0
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.1- Direct-mapped cache: data request

Example:

- Main memory: 256B (addresses: 0 to 255)
- Cache: 32B direct-mapped, 8 * 4B cache lines



	Valid Bit	Tag	Data
0	0	-	-
1	1	0	0x9E768A8A
2	1	2	0x45F36C4D
3	1	1	0xFFB78E14
4	1	3	0xA57B9426
5	0	-	-
6	1	4	0x710F84E3
7	0	-	-

3.1- Direct-mapped cache: conflict misses and thrashing

- Conflict miss rate is high
- Very high conflict miss rate = thrashing

Example:

- Load address 44 in cache entry 3

⇒Compulsory miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
1	0	0x9E768A8A	1
2	2	0x45F36C4D	1
3	1	0xFFB78E14	1
4	3	0xA57B9426	1
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.1- Direct-mapped cache: conflict misses and thrashing

- Conflict miss rate is high
- Very high conflict miss rate = thrashing

Example:

- Load address 44 in cache entry 3
- Load address 77 in cache entry 3, evicts address 44

⇒Compulsory miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
1	0	0x9E768A8A	1
2	2	0x45F36C4D	1
3	2	0x9634E88A	1
4	3	0xA57B9426	1
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.1- Direct-mapped cache: conflict misses and thrashing

- Conflict miss rate is high
- Very high conflict miss rate = thrashing

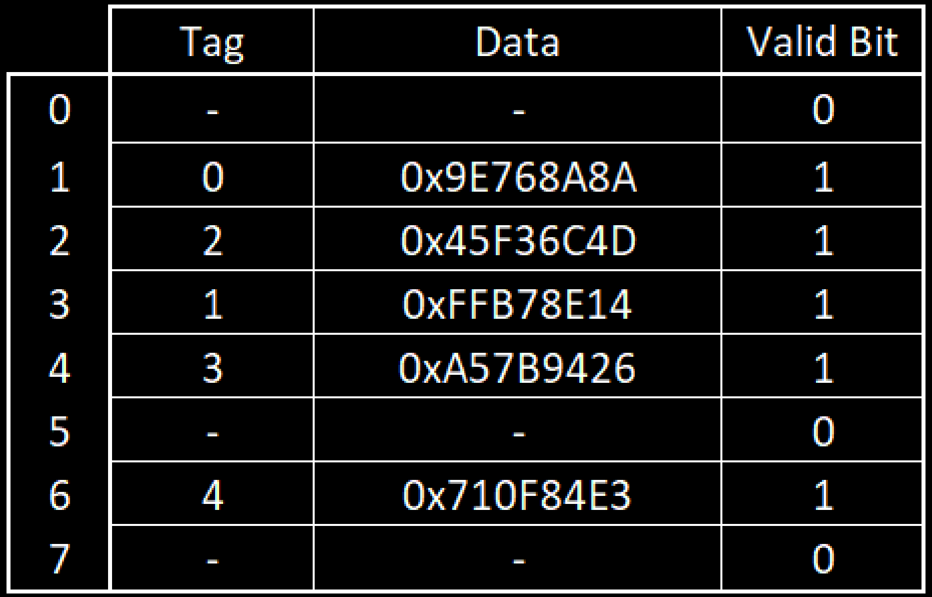
Example:

- Load address 44 in cache entry 3
- Load address 77 in cache entry 3, evicts address 44
- Load address 44 in cache entry 3, evicts address 76

⇒Conflict miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines



	Tag	Data	Valid Bit
0	-	-	0
1	0	0x9E768A8A	1
2	2	0x45F36C4D	1
3	1	0xFFB78E14	1
4	3	0xA57B9426	1
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.1- Direct-mapped cache: conflict misses and thrashing

- Conflict miss rate is high
- Very high conflict miss rate = thrashing

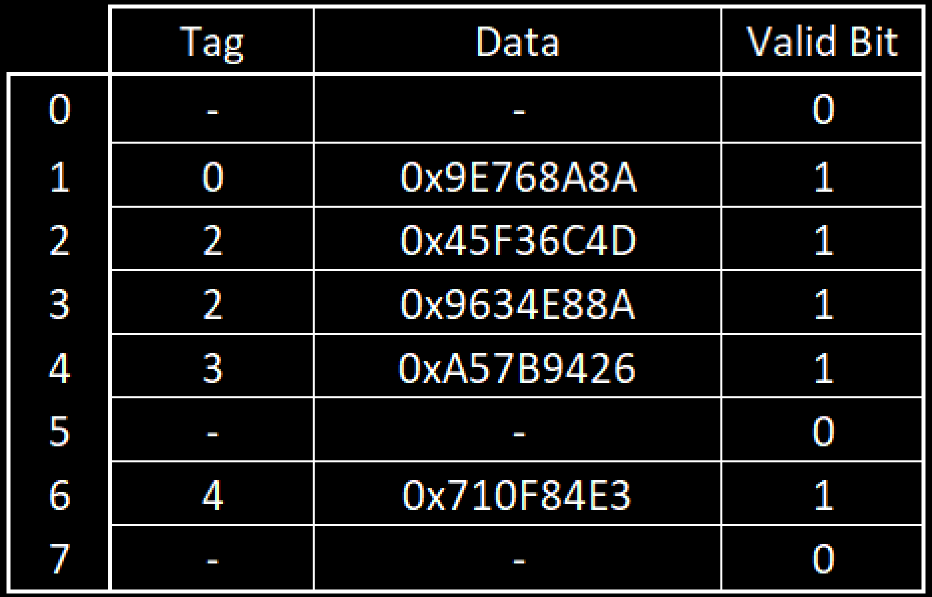
Example:

- Load address 44 in cache entry 3
- Load address 77 in cache entry 3, evicts address 44
- Load address 44 in cache entry 3, evicts address 76
- Load address 77 in cache entry 3, evicts address 44

⇒Conflict miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines



	Tag	Data	Valid Bit
0	-	-	0
1	0	0x9E768A8A	1
2	2	0x45F36C4D	1
3	2	0x9634E88A	1
4	3	0xA57B9426	1
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.1- Direct-mapped cache: conflict misses and thrashing

- Conflict miss rate is high
- Very high conflict miss rate = thrashing

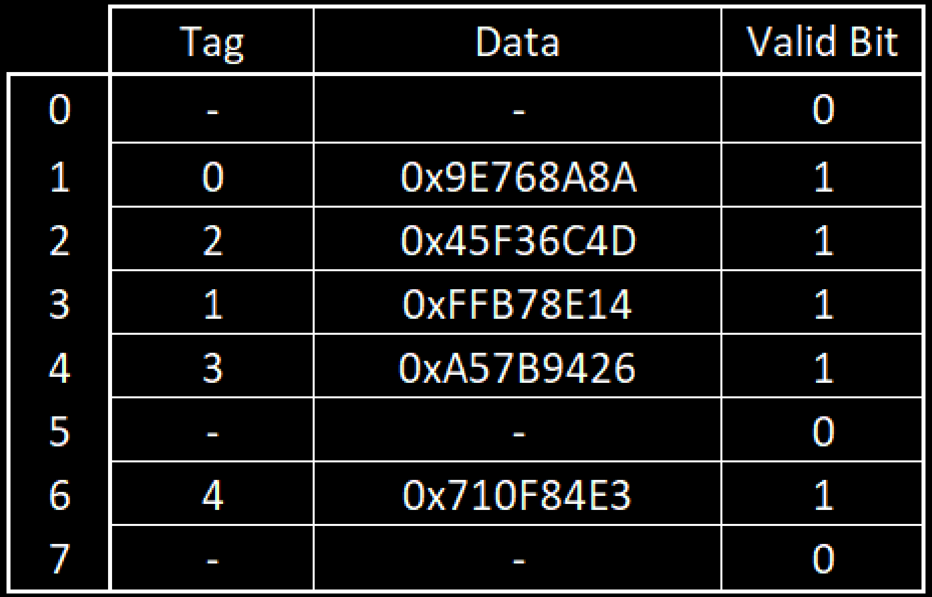
Example:

- Load address 44 in cache entry 3
- Load address 77 in cache entry 3, evicts address 44
- Load address 44 in cache entry 3, evicts address 76
- Load address 77 in cache entry 3, evicts address 44
- …

Conflict miss every time from step 3 onward !

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines

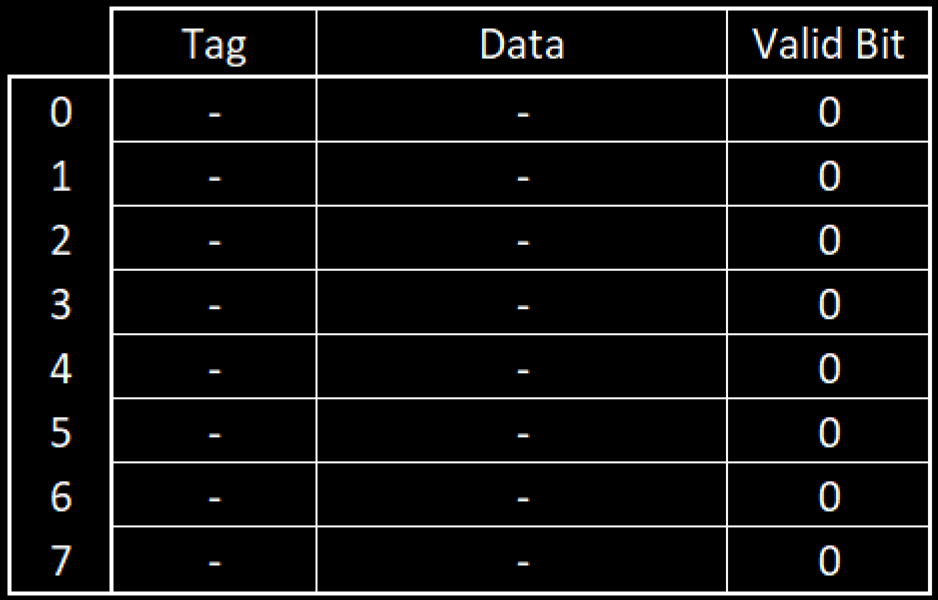


	Tag	Data	Valid Bit
0	-	-	0
1	0	0x9E768A8A	1
2	2	0x45F36C4D	1
3	1	0xFFB78E14	1
4	3	0xA57B9426	1
5	-	-	0
6	4	0x710F84E3	1
7	-	-	0

3.2- Fully associative cache

Example:

- Main memory:
  - 256B (addresses: 0 to 255)
- Cache:
  - 32B fully associative
  - 8 cache lines
  - 4B line size
  - 6b tag



	Tag	Data	Valid Bit
0	-	-	0
1	-	-	0
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.2- Fully associative cache: loading memory

- CPU request data from main memory with address 155.

155 ⇔ 10011011

- Block size is 4B = 2^{2}B, set 2 least significant bits of address to 0 to find block address

10011011 & 11111100 = 10011000 ⇔ 152

- No need to compute block index, find any empty cache entry
- Use the block address as the tag for the entry

Example:

- Main memory: 256B (addresses: 0 to 255)

address	151	152	153	154	155	156
data	5 B	7 1	0 F	8 4	E 3	9 2

$\underbrace{\text{152 153 154 155}}_{\text{memory block}}$

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	152	0x710F84E3	1
1	-	-	0
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

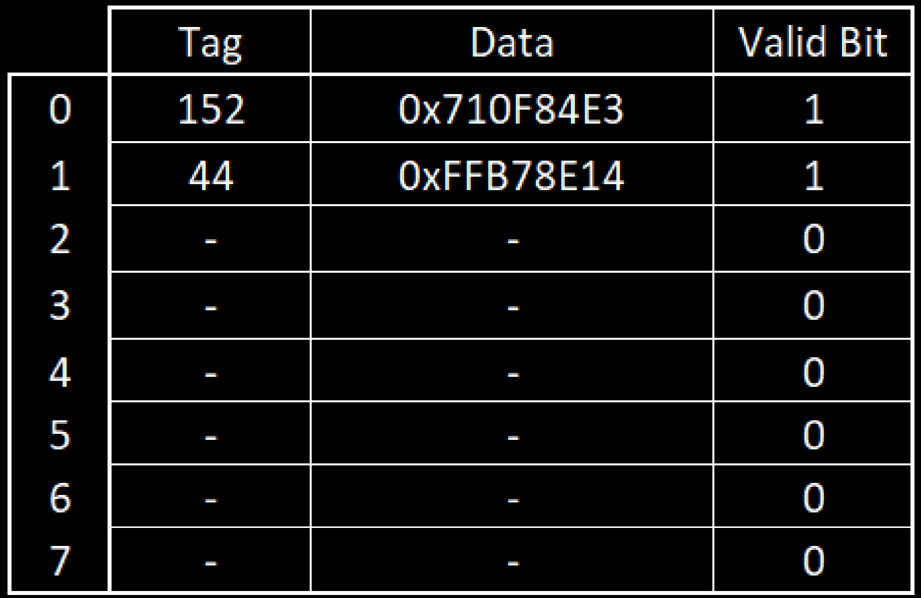
3.2- Fully associative cache: no more conflict misses

- Load address 44 in cache entry 2

⇒Compulsory miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines



	Tag	Data	Valid Bit
0	152	0x710F84E3	1
1	44	0xFFB78E14	1
2	-	-	0
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.2- Fully associative cache: no more conflict misses

- Load address 44 in cache entry 2
- Load address 77 in cache entry 3

⇒Compulsory miss

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	152	0x710F84E3	1
1	44	0xFFB78E14	1
2	76	0x9634E88A	1
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.2- Fully associative cache: no more conflict misses

- Load address 44 in cache entry 2
- Load address 77 in cache entry 3
- Read address 44 in cache entry 2

⇒Cache hit !

Example:

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	152	0x710F84E3	1
1	44	0xFFB78E14	1
2	76	0x9634E88A	1
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.2- Fully associative cache: no more conflict misses

- Load address 44 in cache entry 2
- Load address 77 in cache entry 3
- Read address 44 in cache entry 2
- Read address 77 in cache entry 3

⇒Cache hit !

- No more conflict misses, but …

Example:

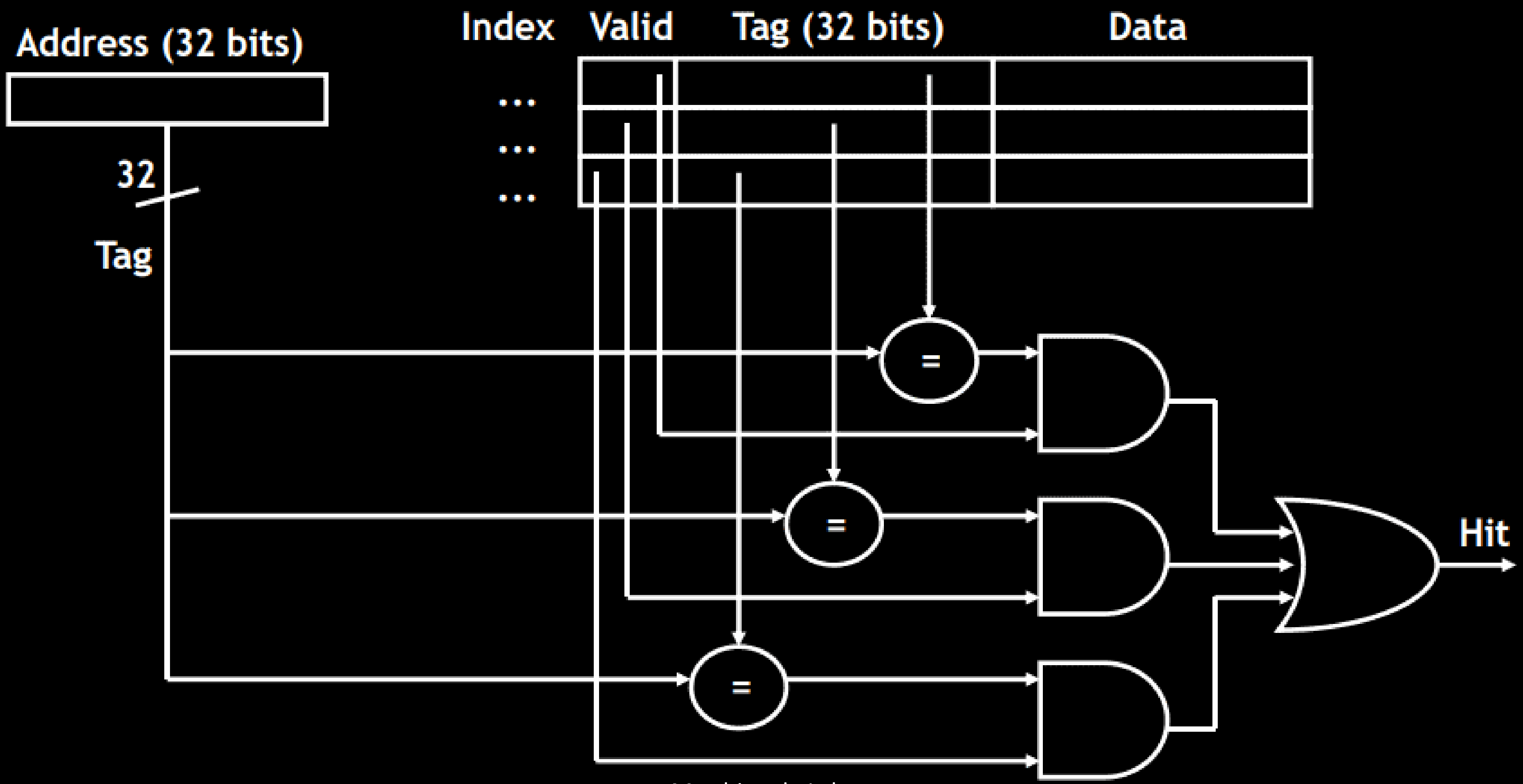
- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	152	0x710F84E3	1
1	44	0xFFB78E14	1
2	76	0x9634E88A	1
3	-	-	0
4	-	-	0
5	-	-	0
6	-	-	0
7	-	-	0

3.2- Fully associative cache: the cost of data request

Example:

- Main memory: 256B (addresses: 0 to 255)
- Cache: 32B direct-mapped, 8 * 4B cache lines

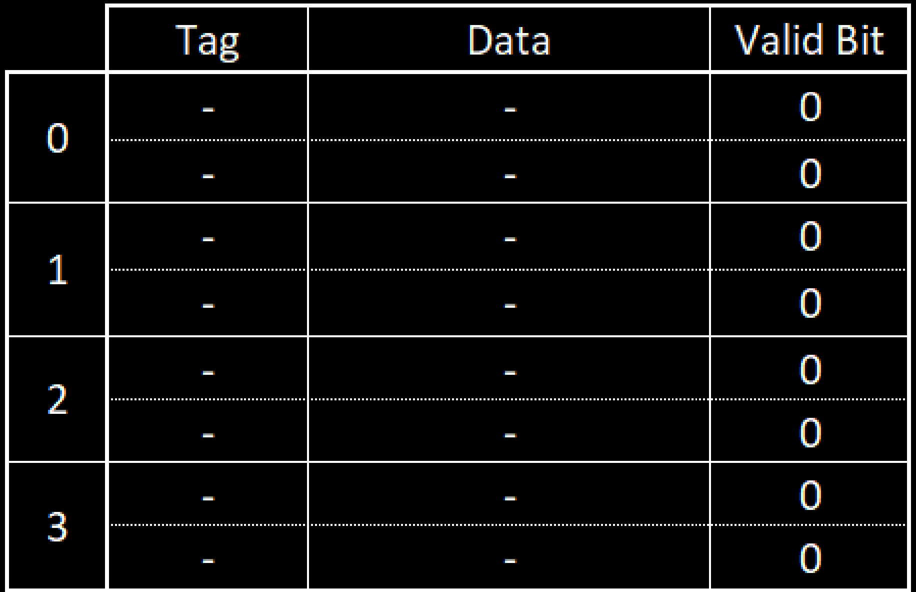


Matthieu de Laharpe

3.2- Set-associative cache

Example:

- Main memory:
  - 256B (addresses: 0 to 255)
- Cache:
  - 32B 2-way associative
  - 4 sets
  - 4B line size
  - 4b tag



	Tag	Data	Valid Bit
0	-	-	0
	-	-	0
1	-	-	0
	-	-	0
2	-	-	0
	-	-	0
3	-	-	0
	-	-	0

3.3- Set-associative cache: loading memory

- CPU request data from main memory with address 155.

155 ⇔ 10011011

- Block size is 4B = 2^2B, set 2 least significant bits of address to 0 to find block address

10011011 & 11111100 = 10011000 ⇔ 152

- Right shift block address by 2 to find block index in main memory (equ. to division by 4)

10011000 >> 2 = 00100110 ⇔ 38

Example:

- Main memory: 256B (addresses: 0 to 255)

address	151	152	153	154	155	156
data	5 B	7 1	0 F	8 4	E 3	9 2

memory block

- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
	-	-	0
1	-	-	0
	-	-	0
2	-	-	0
	-	-	0
3	-	-	0
	-	-	0

3.3- Set-associative cache: loading memory

Example:

- Main memory: 256B (addresses: 0 to 255)

address	151	152	153	154	155	156
data	5 B	7 1	0 F	8 4	E 3	9 2

memory block

- There are 4 = 2^2 cache sets, keep the 2 least significand bits of that index to find the set index (equ. to mod 4)

00100110 & 00000011 = 00000010 ⇔ 2

- Find an empty cache entry in the set
- Use the remaining bits as the tag.

(00100110 & 11111100) >> 2 = 00001001 ⇔ 9

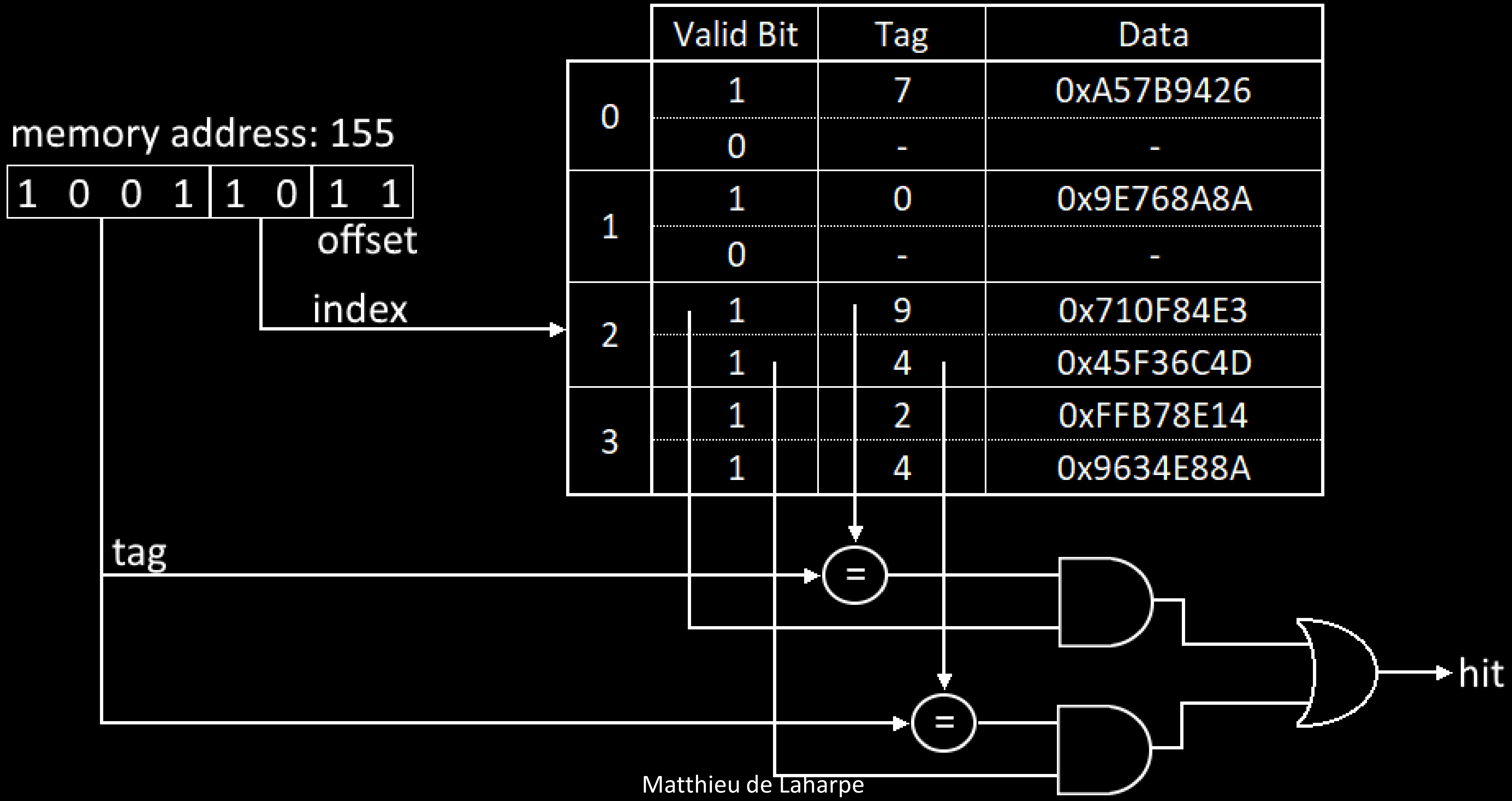
- Cache: 32B direct-mapped, 8 * 4B cache lines

	Tag	Data	Valid Bit
0	-	-	0
	-	-	0
1	-	-	0
	-	-	0
2	9	0x710F84E3	1
	-	-	0
3	-	-	0
	-	-	0

3.3- Set-associative cache: balanced cost for data requests

Example:

- Main memory: 256B (addresses: 0 to 255)
- Cache: 32B direct-mapped, 8 * 4B cache lines



memory address: 155

	Valid Bit	Tag	Data
0	1	7	0xA57B9426
	0	-	-
1	1	0	0x9E768A8A
	0	-	-
2	1	9	0x710F84E3
	1	4	0x45F36C4D
3	1	2	0xFFB78E14
	1	4	0x9634E88A

4- Cache policies

- CPU cache is a critical component regarding performances

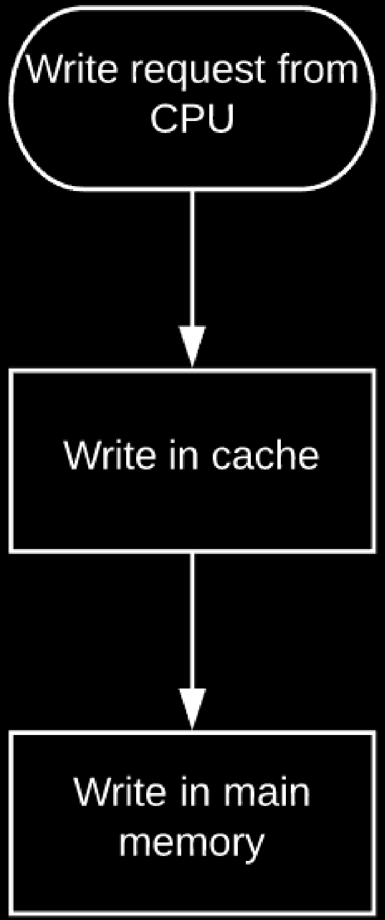
L1 cache ref	0,5 ns	
L2 cache ref	7 ns	14 * L1 cache
Main memory ref	100 ns	14 * L2 cache, 200 * L1 cache

- CPU cache behaves differently depending on its characteristics, environment and on the applications that use it

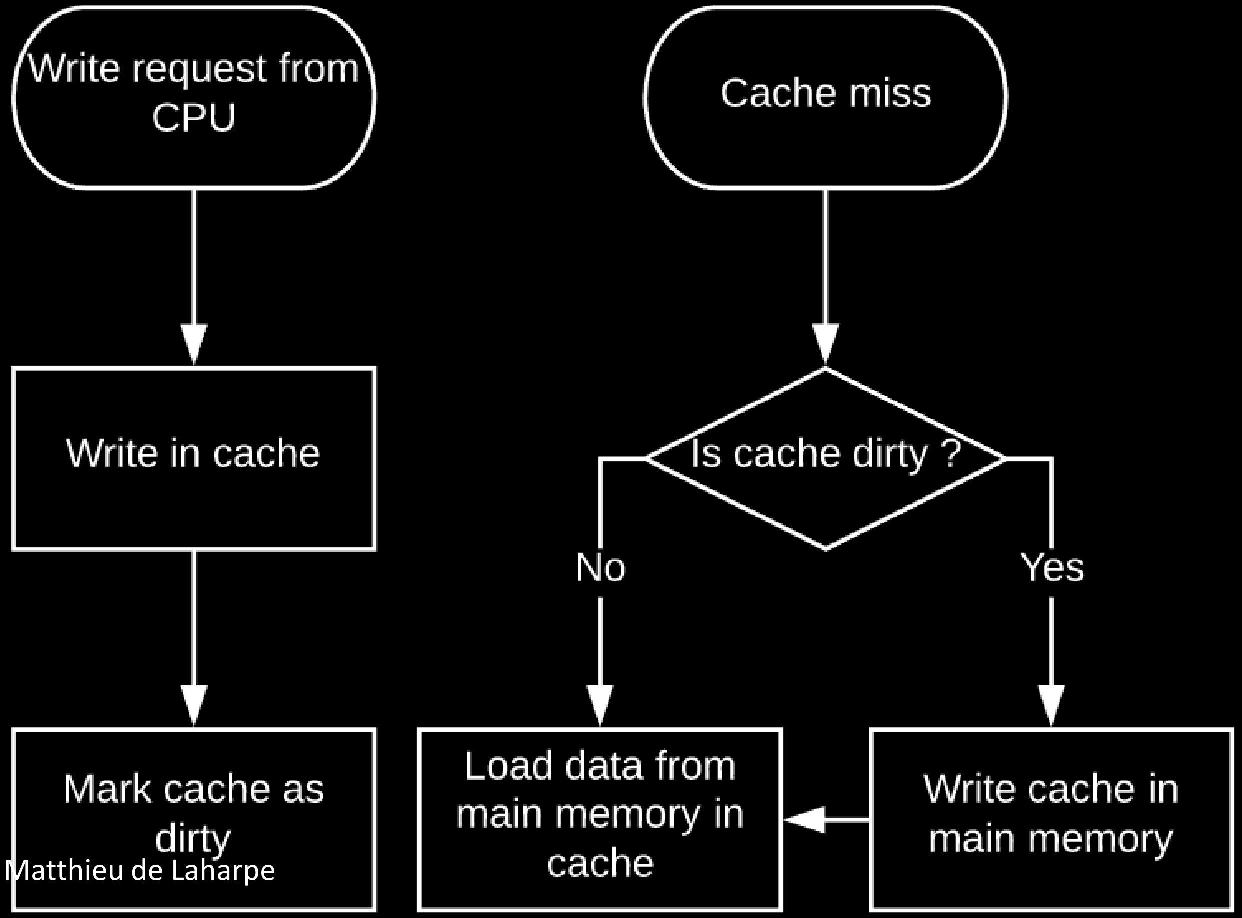
- A lot of different policies have been defined to dictate cache behavior and adapt to different situations

4.1- Write policies:

Write through



Write back (requires a dirty flag in the cache entry)



4.1- Write policies

When a write triggers a cache miss:

- Write allocate: load data at the missed location to the cache
- No-write allocate: write directly to the main memory/next level cache

Policies are usually paired in this way:

- Write back + write allocate -> hoping for subsequent w/r to the same location
- Write through + no write allocate

Intermediate write through policy:

- Write operations are buffered together to save time, power and bus utilization
- AMD Bulldozer architecture uses a specialised Write Coalescing Cache between L1d and L2

4.2- Replacement policies

Which cache entry should be evicted to make room for a new cache line ?

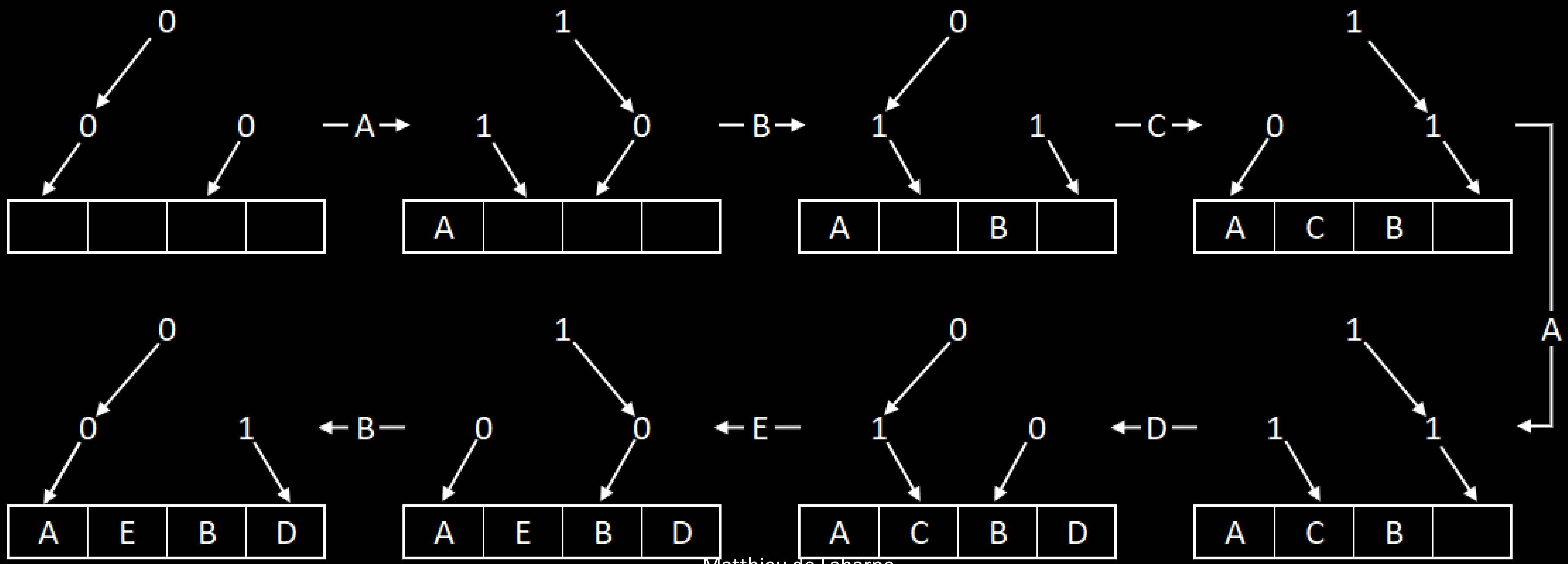
- Wide variety of policies
  - Least Recently Used (LRU) -> temporal locality, very easy on 2-way associative cache (1bit)
  - Most Recently Used (MRU) -> useful when sequentially looping over a data set
  - First In first Out (FIFO) -> simplest LRU
  - Least Frequently Used (LFU) -> can lead to cache pollution
  - Random
  - …
- A lot of implementations are variations, improvements or hybrids of LRU
  - LRFU, EELRU, LRU-K, LRU-2 …
  - Tree Pseudo LRU, bit Pseudo LRU …

4.2- Replacement policies: Pseudo LRU tree

- Requires N-1 bits per set for a N-way set-associative cache
  - On a 256KiB 4-way associative cache with a 64B line size: (L2 cache in Intel Kaby Lake µarch)

    256KiB / (4*64B) = 1024 sets

    3b*1024 = 3Kib = 384B = 0,375KiB (≈0,15% of cache size)



5- Cache coherence

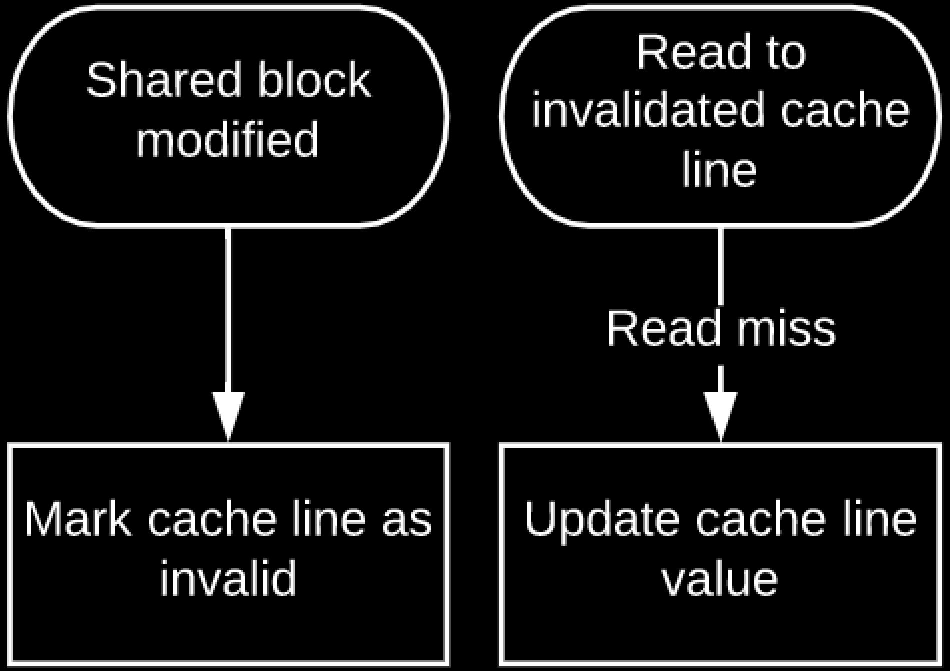
- Maintain uniformity of shared data resources across multiple local caches in multi CPU system

- When a CPU modifies the value of a shared data, this modification must be propagated to all the other caches

- Two most common mechanisms:
  - Bus snooping: cache have a coherency controller that monitors the bus for changes in shared memory blocks
  - Directory-based

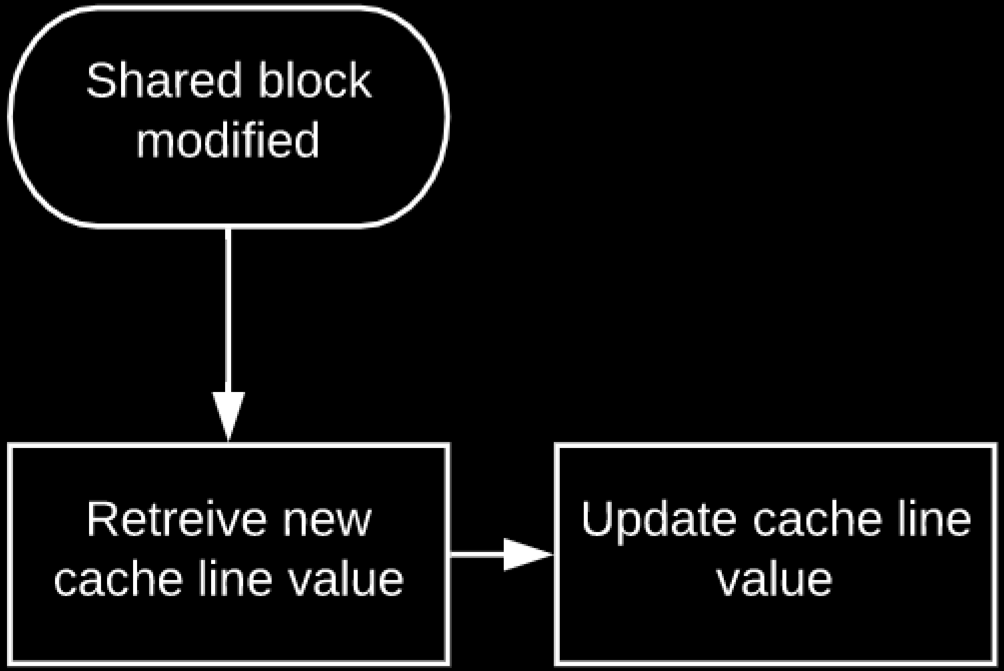
5- Cache coherence: bus snooping

Write Invalidate



Write Update

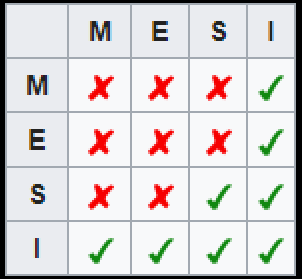
- Higher bus usage than invalidate



5- Cache coherence: MESI protocol

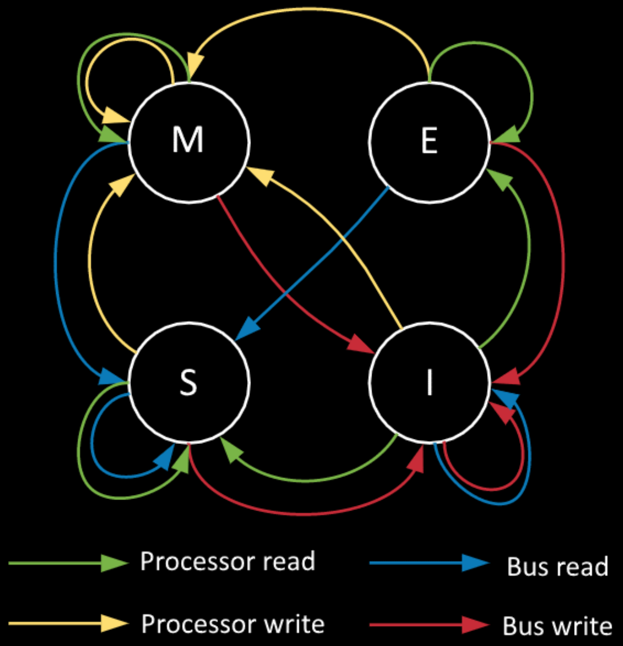
- Cache line can be in 4 different states:
  - Modified: cache line is present in 1 cache, doesn’t match the backing store
  - Exclusive: cache line is present in 1 cache, matches the backing store
  - Shared: cache line is present in multiple caches, matches the backing store
  - Invalid: cache line is invalid

Relative state of cache lines with the MESI protocol



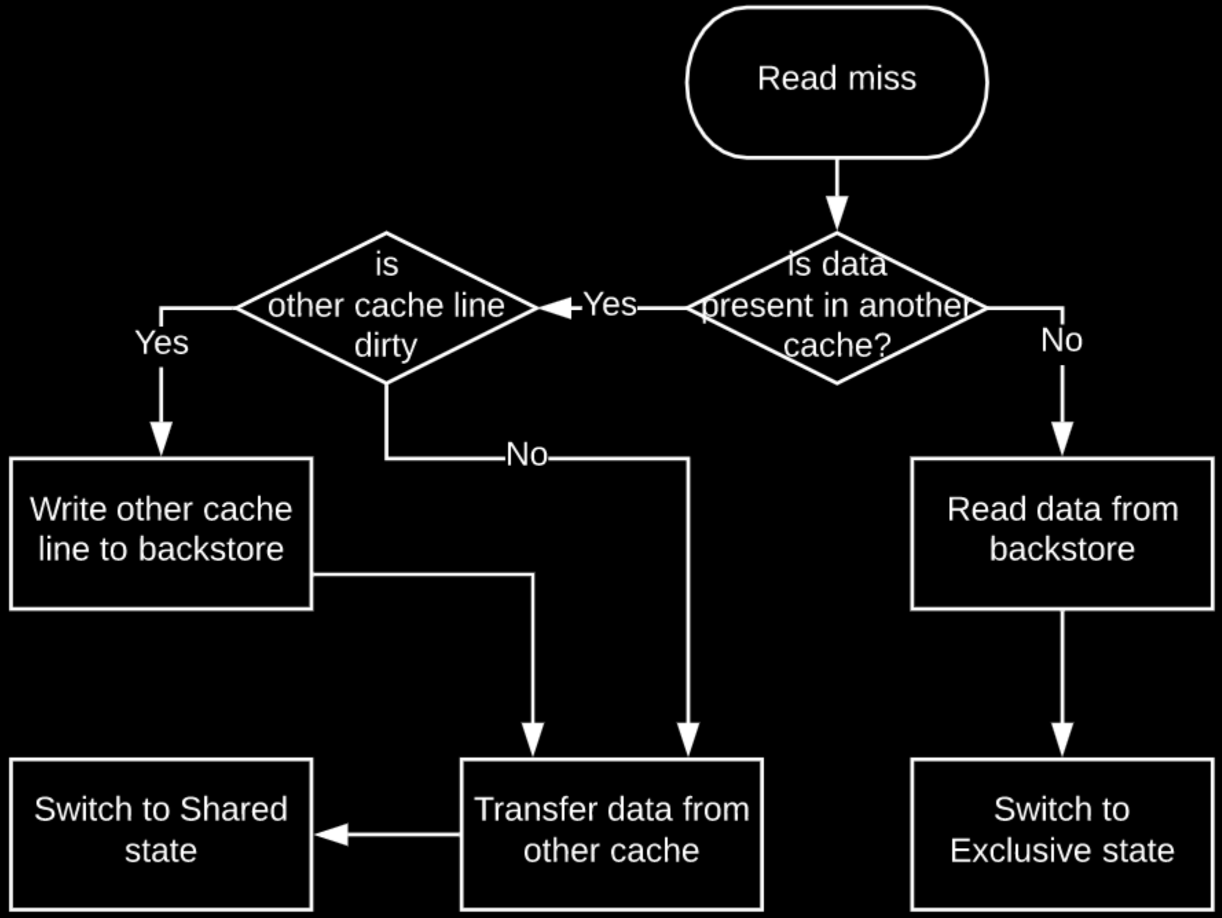
	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

MESI state transitions



Matthieu de Laharpe

5- Cache coherence: MESI protocol



Read miss
is data present in another cache?
Yes
No
is other cache line dirty
Yes
No
Write other cache line to backstore
Read data from backstore
Switch to Shared state
Transfer data from other cache
Switch to Exclusive state

6- Going further

- Address translation
  - Modern systems use virtual memory
  - Need translation between virtual and physical addresses
  - Specialized caches handle address translations: Translation Lookaside Buffer
- Other specialized caches
  - Victim cache: holds evicted cache lines to reduce conflict misses
  - Trace cache, micro-operation cache …
- Cache pollution
- Cache control instructions
- Branche prediction and cache prefetching

References

Wikipedia

- Cache (computing)
- CPU cache
- Cache coherence
- Cache replacement policies
- Pseudo LRU
- Cache pollution
- Cache performance
- Thrashing (computer science)
- Cache prefetching
- Bus snooping
- MESI protocol

University of Washington, Computer Science & Engineering lectures from winter 2011:

- Lecture 16: Introduction to caches
- Lecture 17: Locality & Cache organization
- Lecture 18: More on cache organization
- Lecture 19-20: Cache writing

wikichip.org: i7-7700HQ
wikichip.org: Kaby Lake micro arch Memory Hierarchy

Northeastern University: cache basics
Carnegie Mellon University: cache associativity
University of Wisconsin-Madison: Replacement policies

lwn.net: CPU Caches

MIT Computer Science and AI Lab:

- Adaptive Insertion Policies for High Performance Caching
- Modeling Cache Performance Beyond LRU

General Adaptive Replacement Policies

Latency numbers every programmer should know

The memory wall fallacy