Data Model
in
Memory & Cache

By 执少（顾汉杰）
阿里云Nas文件存储团队

Agenda

- What is data model
- Principle of locality
- Data layer on system
- Data path on x86 architecture
- History of cache chips
- Memory & Cache Hierarchy
- How does cache memory work

Data Model in different storage

- The layout of data in memory,

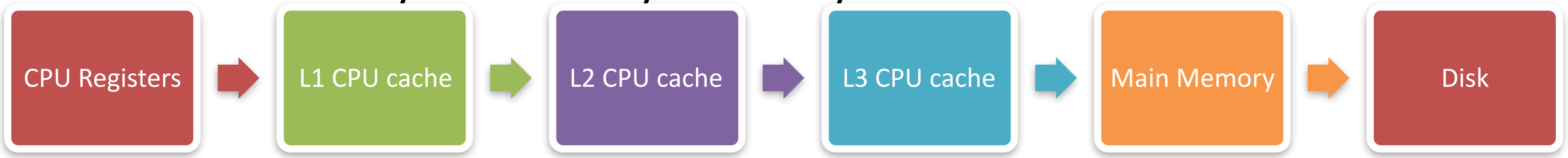
  that is, **Data Structure**
- The layout of data in file,

  that is, **File Format**
- The layout of data in network,

  that is, **Protocol**

Principle of locality

- Programs tend to reuse data and instructions they have used recently, or related storage locations, are frequently accessed.
- Types
  - temporal locality
  - spatial locality
- Optimization techs
  - caching
  - prefetching
- Relevant factors
  - Structure of program
  - Linear data structure
  - Use efficiency for memory hierarchy



Data Layer

- <u>Core in Processor(load instructions & data)</u>
- <u>Cache(L1, L2, or L3, blocked by "cache line")</u>
- <u>Main memory(blocked by "page")</u>
- <u>File system(blocked by "file block")</u>
- <u>Disk(blocked by "sector")</u>

CPU Stalls

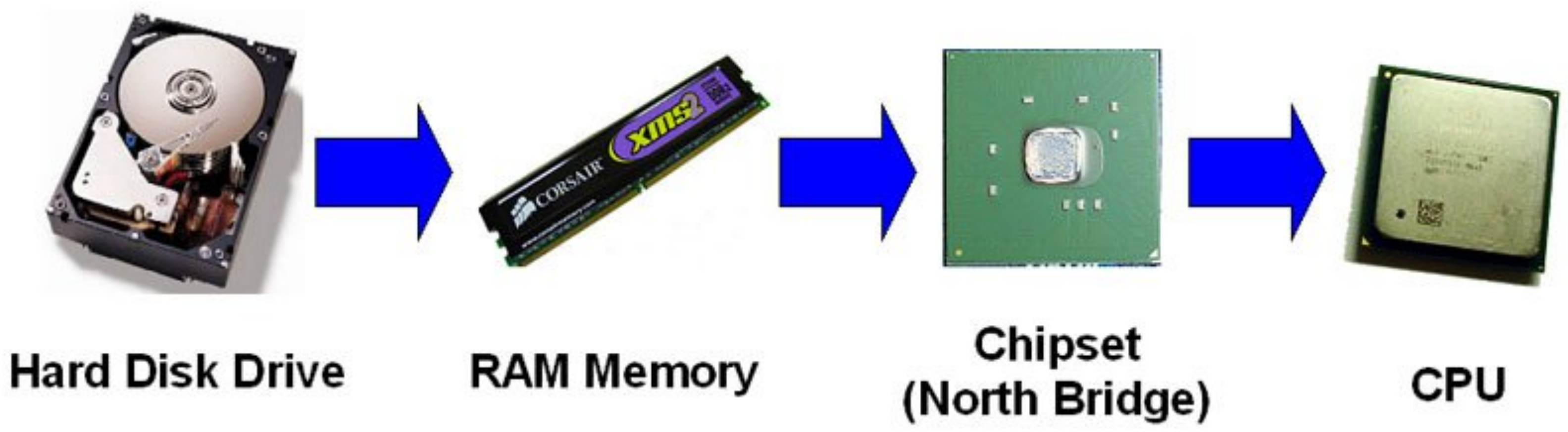
Cache hit/miss

Swap in/out

IO await

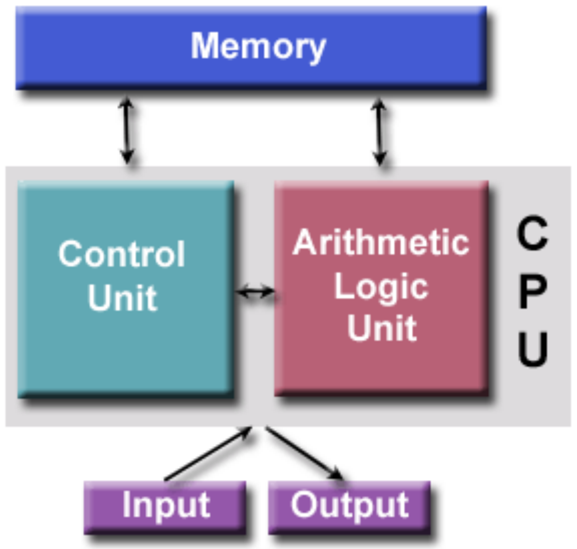
Data path on x86

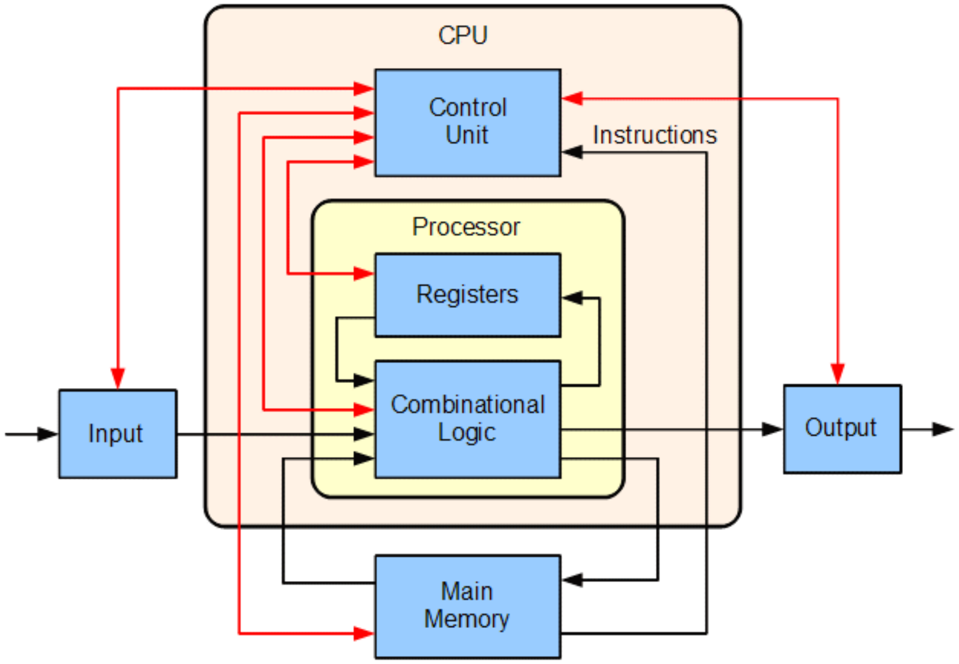
Transfer rates of all components differ, that's the ***question****!*



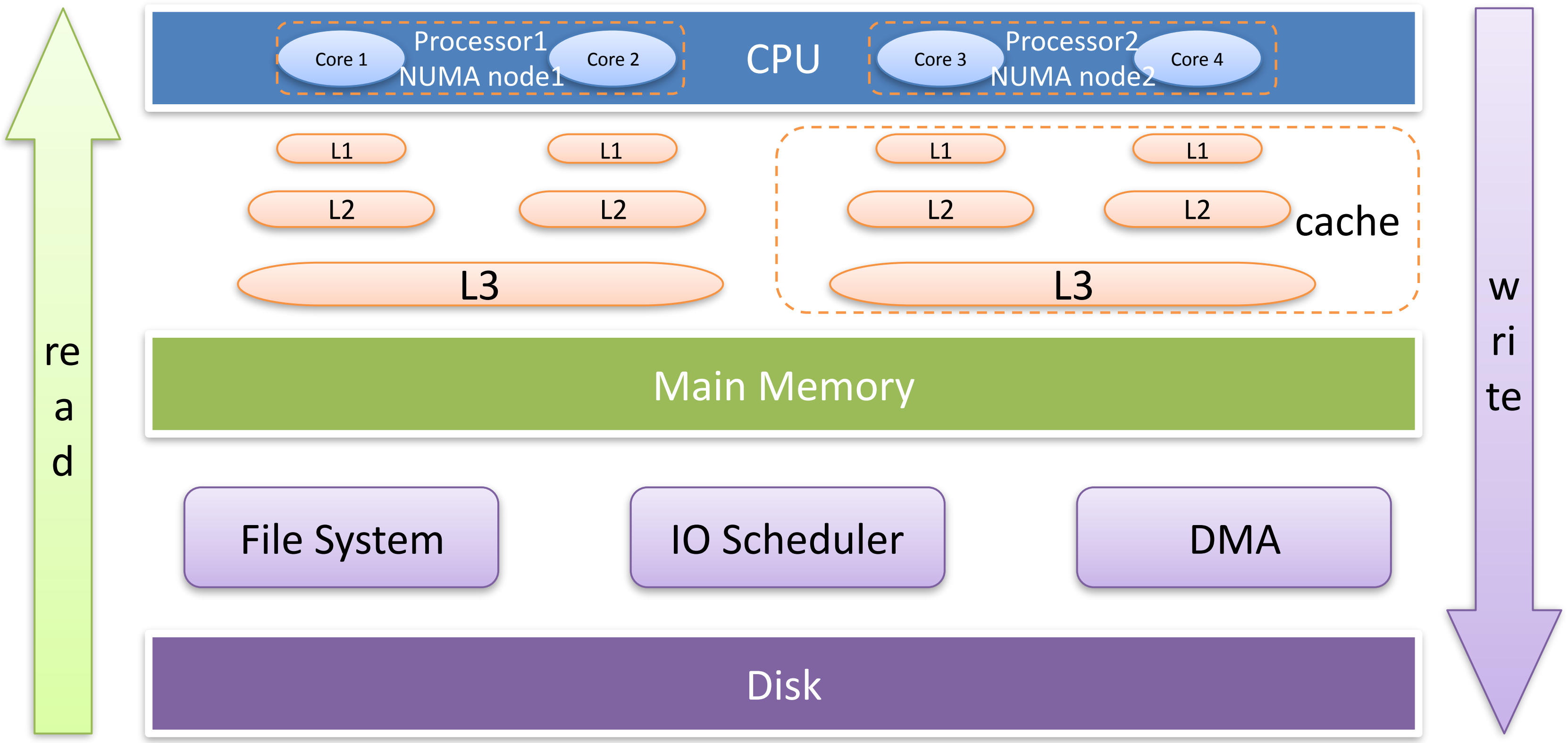
Computer Architecture

- **von Neumann Architecture**





Data Path on x86/x64



History of cache on chips

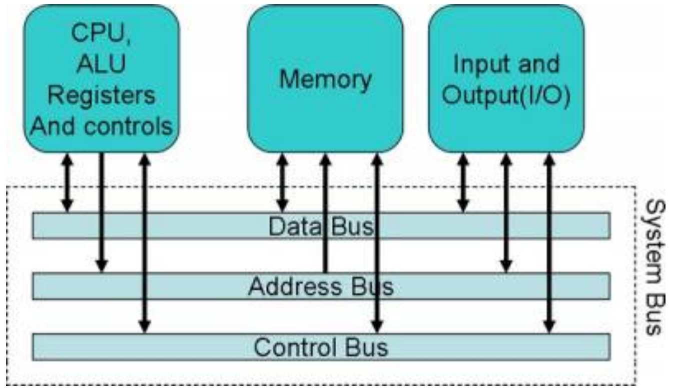
- **Intel 8086, 8088** (1978)

  First x86, original 4.77Mhz, no cache.

  CPU accesses the memory directly.

  Read path:

  1. The CPU puts the address it want to read on the memory bus and assert the read flag.
  2. Memory puts the data on the data bus.
  3. The CPU copies the data from the data bus to its internal registers.



History of cache on chips

- **80286** (1982)

  6-25Mhz, still no cache.

  The speed of memory access started to be slower than core. Delay occurred between core and memory (as core 20Mhz).

  Read path:

  1. The CPU puts the address on the memory bus and assert the read flag.
  2. Memory starts to put the data on the data bus. The CPU waits
  3. Memory finished getting the data and it is now stable on the data bus
  4. The CPU copies the data from the data bus to its internal registers.

Speed between memory & core mismatched.
That is why we would have cache…

History of cache on chips

- **80386** (1985)

  12-40Mhz, L1 cache on motherboard.

  Core runs faster,

  RAM gets faster, but not as much faster as CPUs.

  Read path:

  1. Check if the data is already in the cache.
  2. If cache hit, read from the much faster cache.
  3. Else, go as 80286.

History of cache on chips

- **80486** (1989)

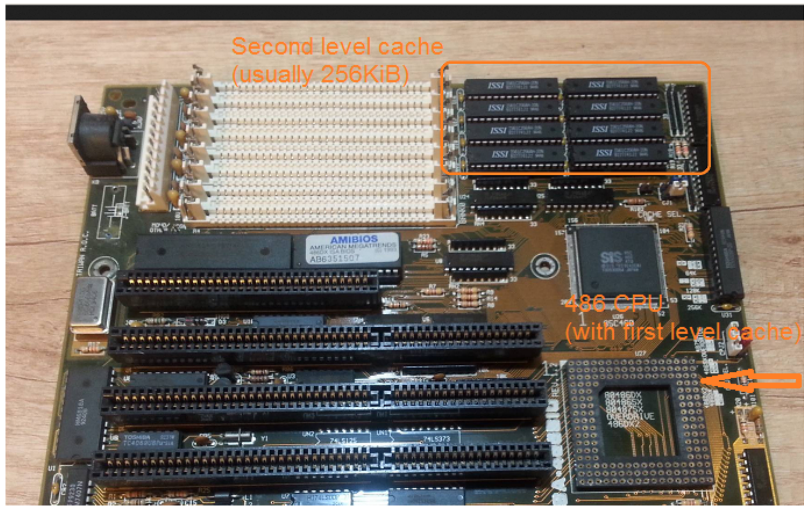
  16-150Mhz, L1 cache on CPU & L2 added.

  A 8KB unified cache used for data and instructions.

  L1 cache on CPU, L2 cache on motherboard.

  Read path:

  1. L1 cache.
  2. L2 cache.
  3. Main memory.



History of cache on chips

- **80586/Pentium** (1993)

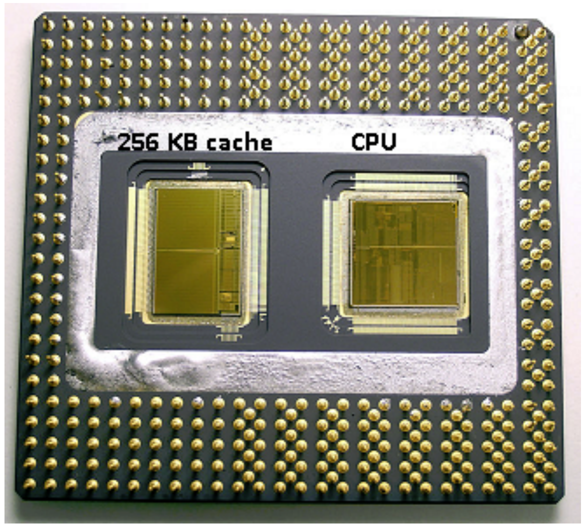
  60-200Mhz, L1 cache was split(L1d & L1i).

  L1: 8 KB each for data and instructions for individually use.

  Chip contains two dies. One with the actual core and L1 cache, and a second die with 256KB L2 cache.

  Read path:

  1. L1 cache(split by instru & data).
  2. L2 cache.
  3. Main memory.



History of cache on chips

- **Pentium 2** (1997)

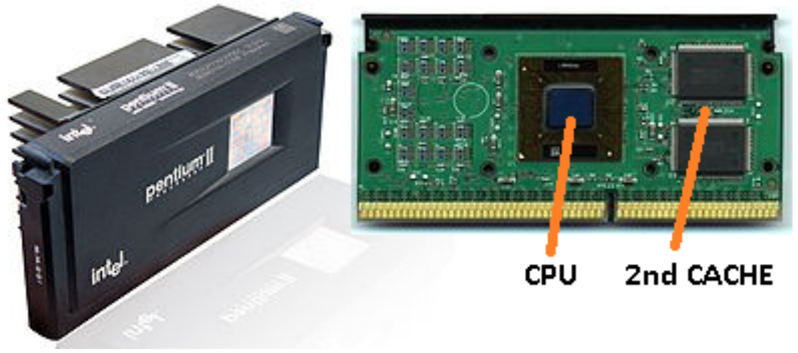
  66-100Mhz, L2 cache no more on chip.

  Components on chip became smaller, financially.

  With one L1 cache per CPU core and a larger but slower L2 cache next to the core.

  Read path:

  1. L1 cache(split by instru & data).
  2. L2 cache.
  3. Main memory.



History of cache on chips

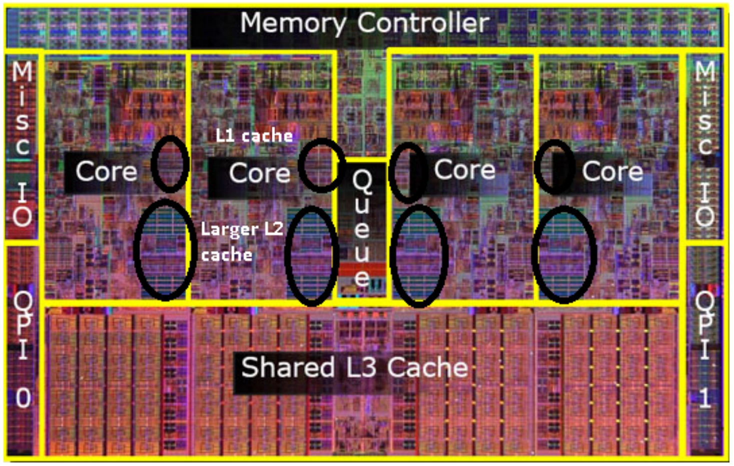
- **Pentium 4** (2000) **& later**

  1.3-3.8Ghz, L3 cache added.

  Reach a limit on single CPU's clock.

  How to solve (3 ways):

  1. Make the CPUs more efficient,
     so they do more work at the same speed.
  2. Use multiple CPUs(multi chips), UMA & SMP.
  3. Use multiple CPUs in the same 'chip', NUMA.



A "dual core" CPU occurred, two or more separate CPU cores are build into a single chip.

"NUMA" architecture became famous.

L3 cache(larger and slower) shared with all CPU cores

An Example

- Intel Intel® Core™ i7 Processor
  - A 32-KB instruction and 32-KB data first-level cache (L1) for each core
  - A 256-KB shared instruction/data second-level cache (L2) for each core
  - 8-MB shared instruction/data last-level cache (L3), shared among all cores

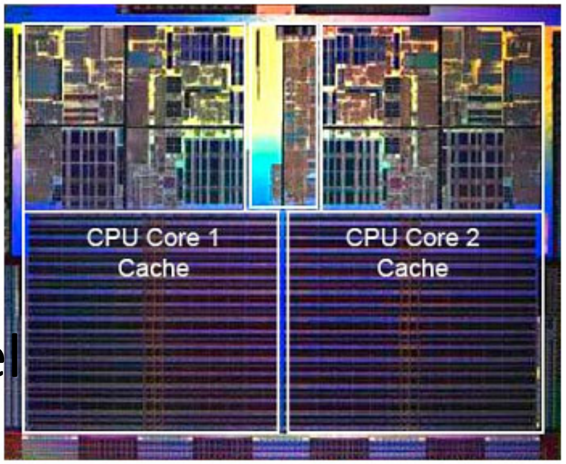
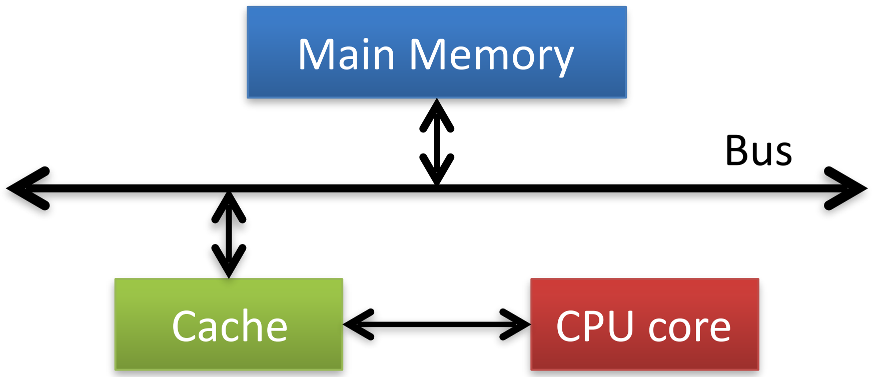


Photo of processor chip.
Cache takes significant area on chip

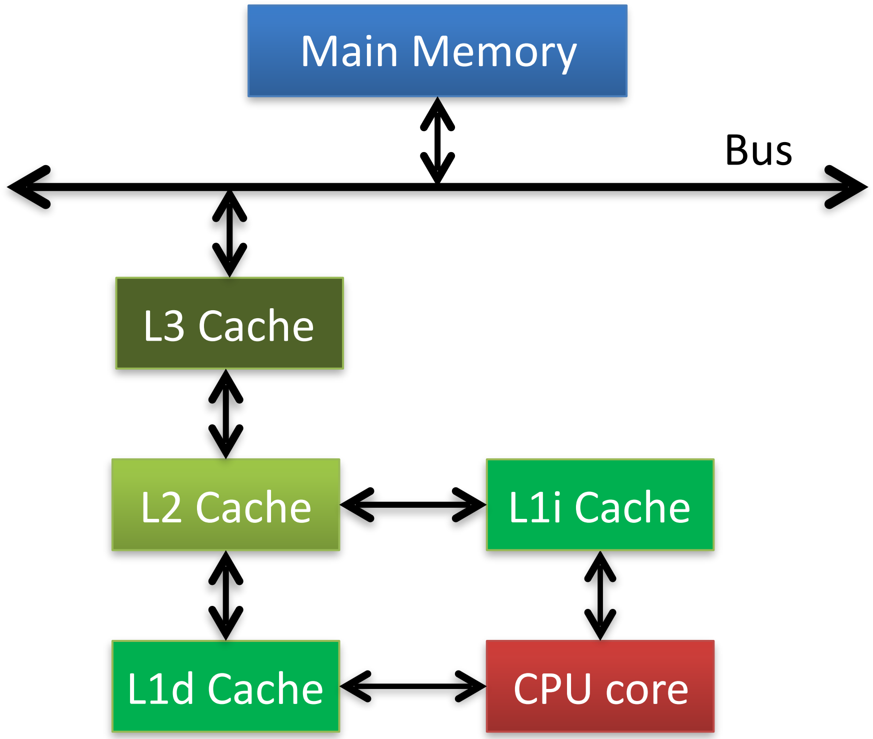
Cache Hierarchy

- MM → cache → (register) → core



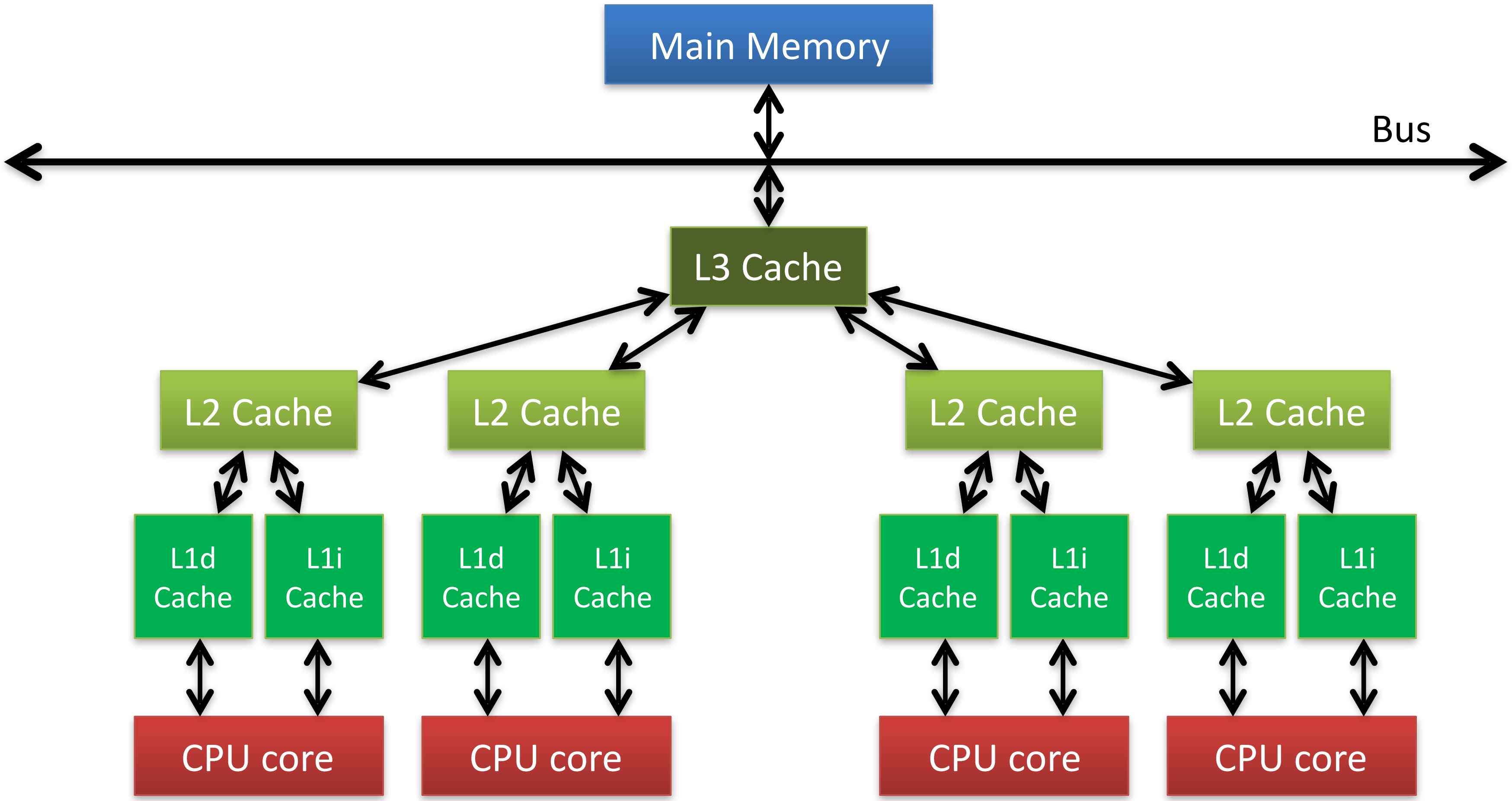
Cache Hierarchy

- MM → L3 → L2 → L1d, L1i → core

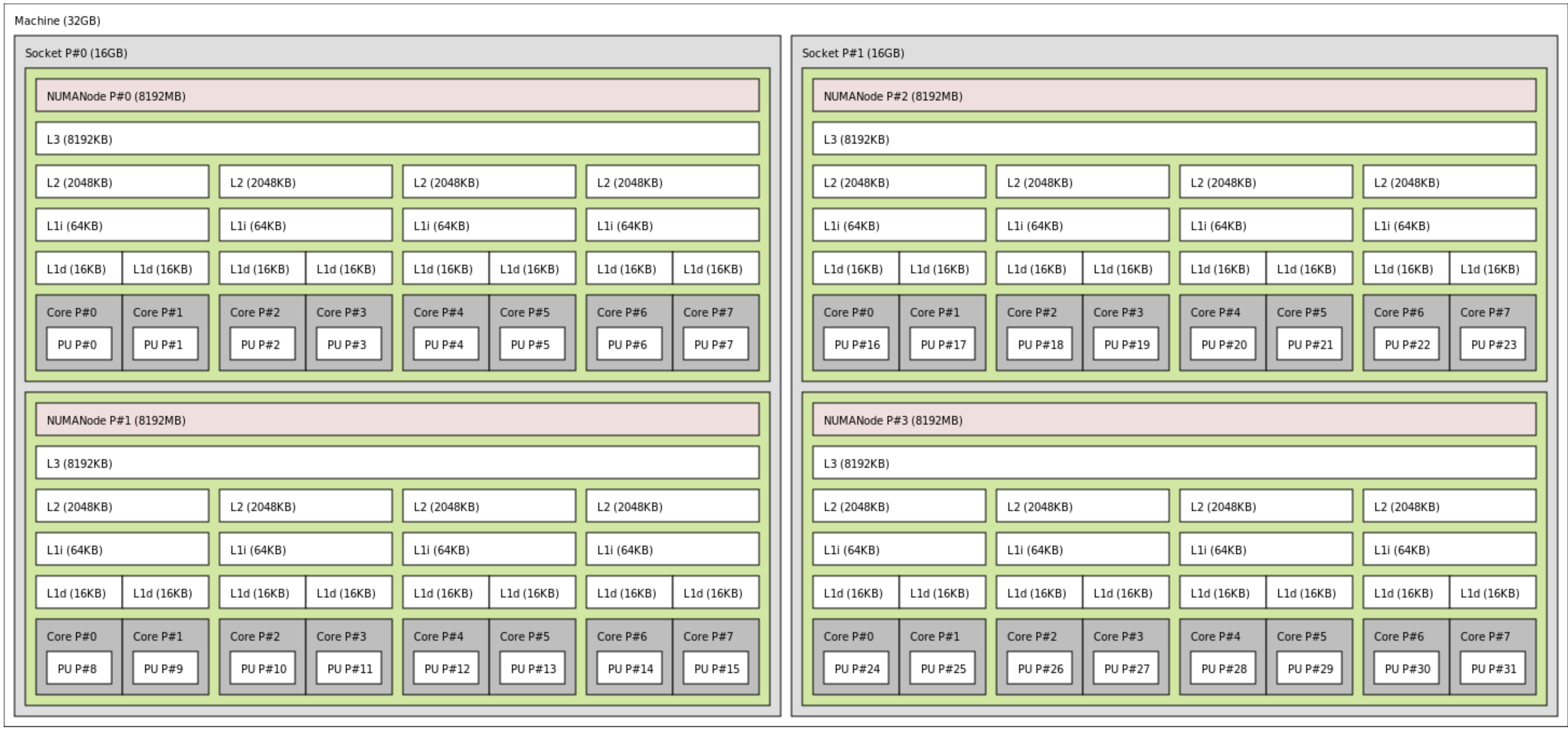


Cache Hierarchy

- UMA, SMP(Symmetric multiprocessing)

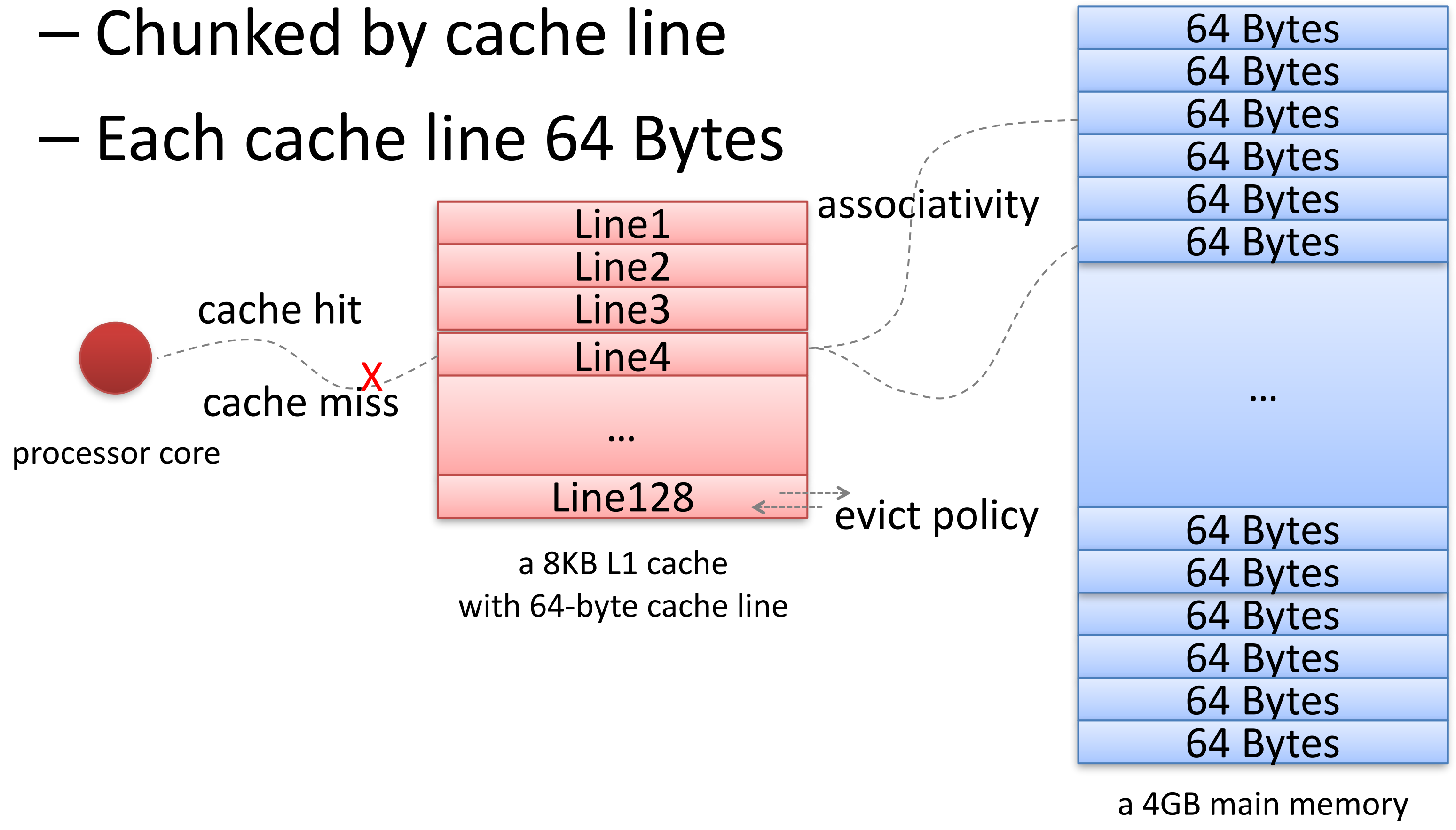


Memory hierarchy of an AMD Bulldozer server



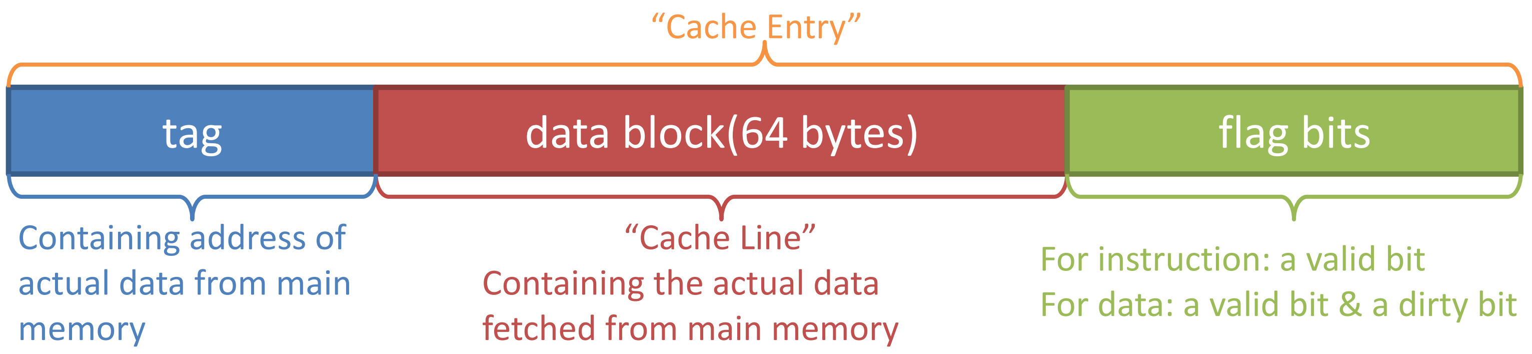
How the cache memory works

- Memory Cache Organization
  - Chunked by cache line
  - Each cache line 64 Bytes

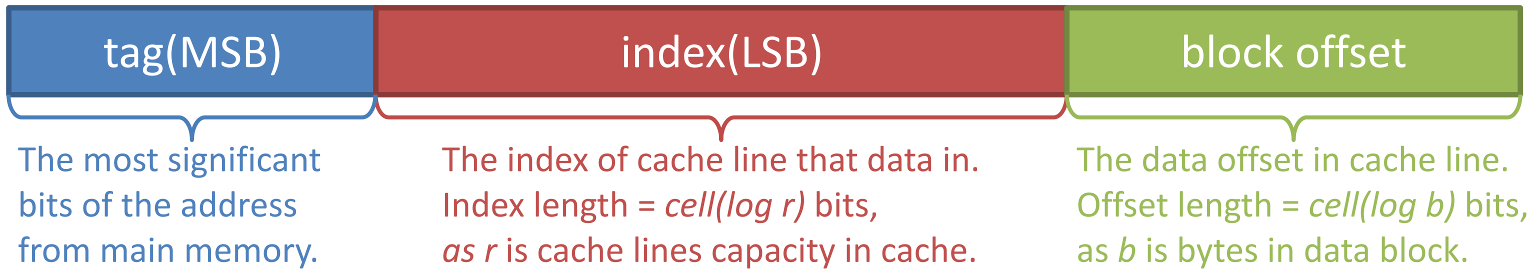


How the cache memory works

- Cache structure


"Cache Entry"
tag
data block(64 bytes)
flag bits
Containing address of actual data from main memory
"Cache Line"
Containing the actual data fetched from main memory
For instruction: a valid bit
For data: a valid bit & a dirty bit

- Cache address


tag(MSB)
index(LSB)
block offset
The most significant bits of the address from main memory.
The index of cache line that data in. Index length = *cell(log r)* bits, *as r* is cache lines capacity in cache.
The data offset in cache line. Offset length = *cell(log b)* bits, as *b* is bytes in data block.

An example

- Pentium 4 processor
  - 4-way set associative L1 cache of 8KB
  - 64-byte size of cache line
  - 32 bits address bus for CPU

→

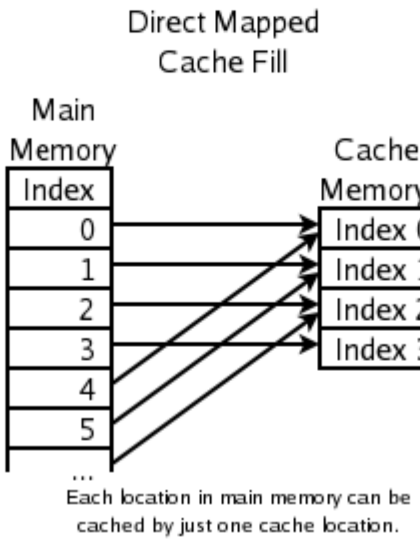
  - 8KB capacity / 64 unit size = 128 cache lines
  - 128 lines / 4 way = 32 lines per way
  - 32 - 5 - 6 = 21

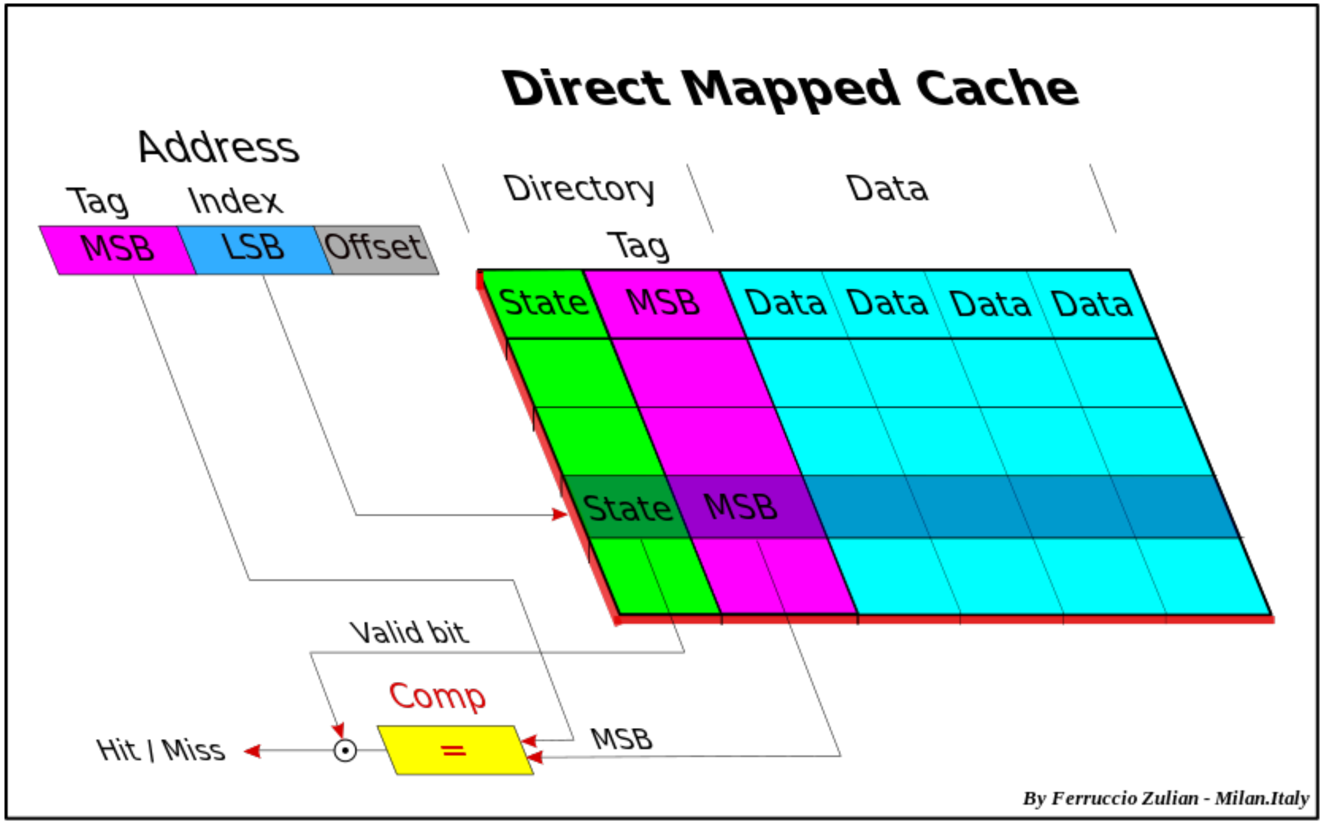
tag	index	block offset
21	5	6

Cache address(32 bits)

How the cache memory works

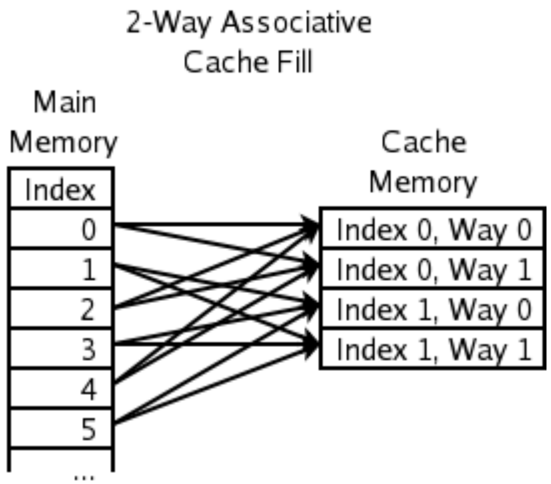
- **Direct mapped cache**
  - Any memory block can be stored in one specific cache entry only.



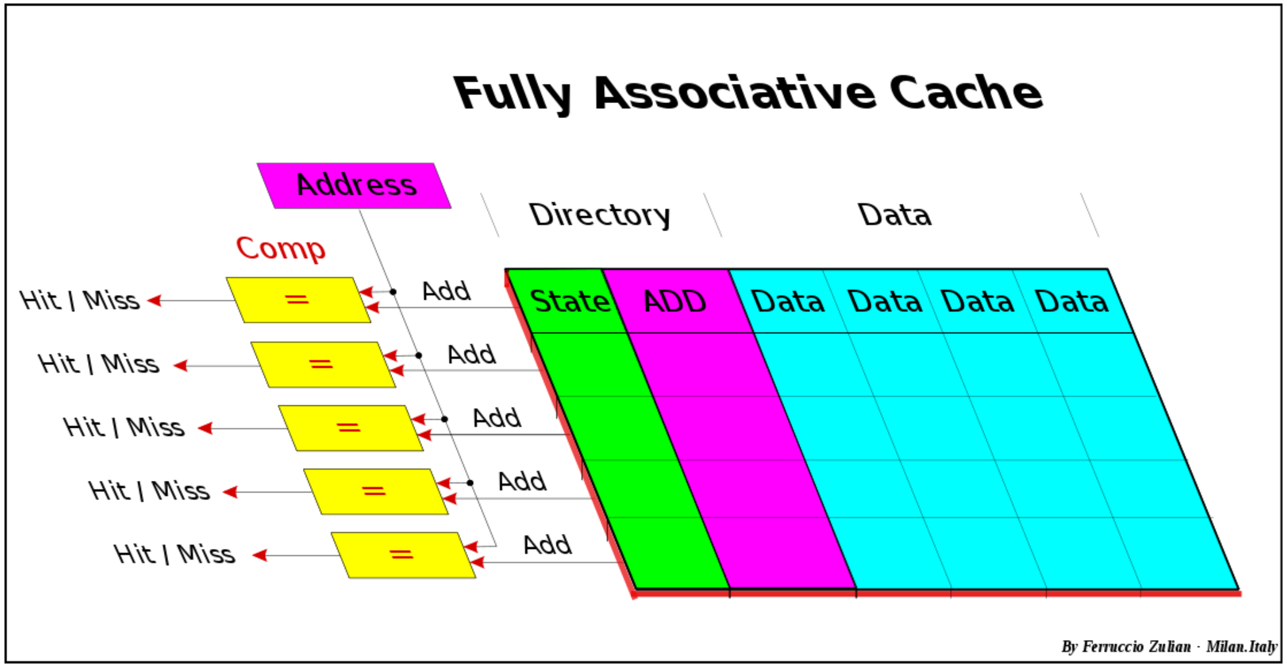


How the cache memory works

- **Fully associative cache**
  - Any memory block can be stored in any cache location.
  - Very small and efficient, just used for TLB.

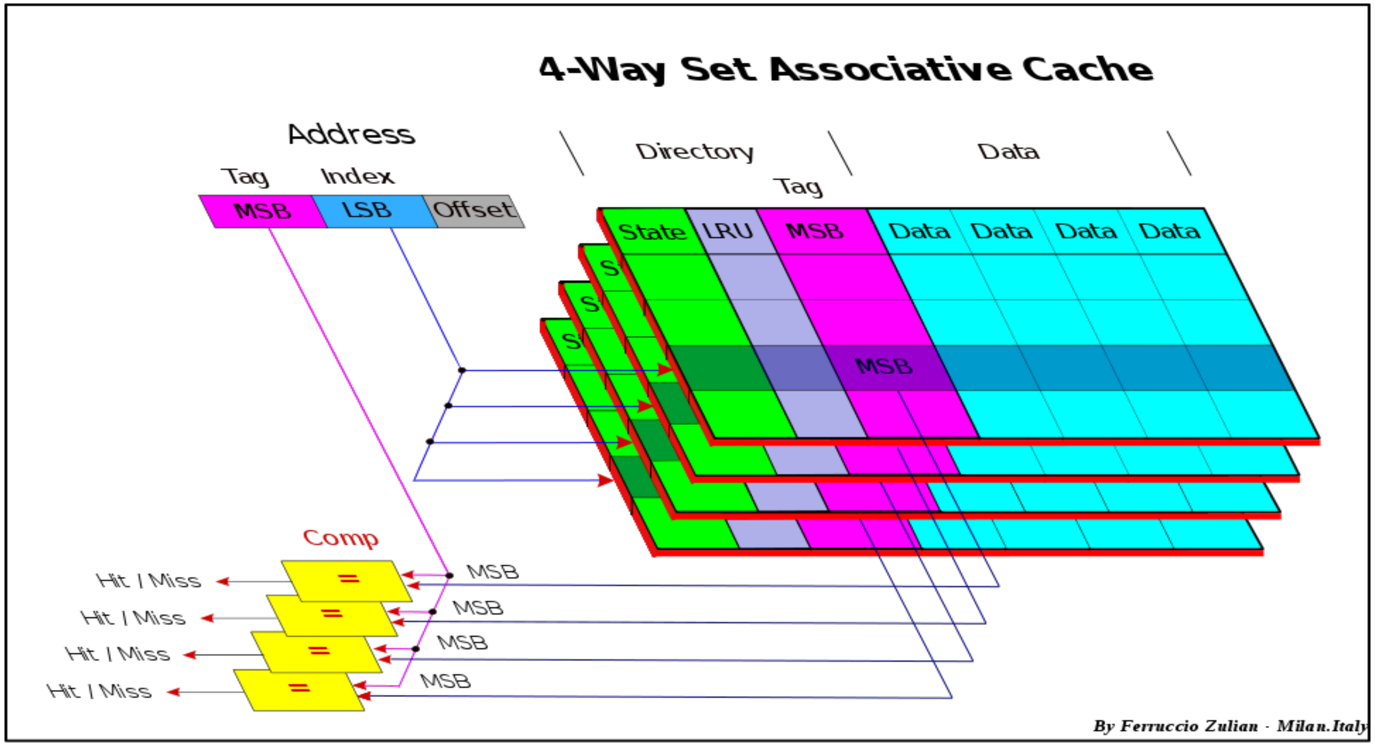


Each location in main memory can be cached by one of two cache locations.



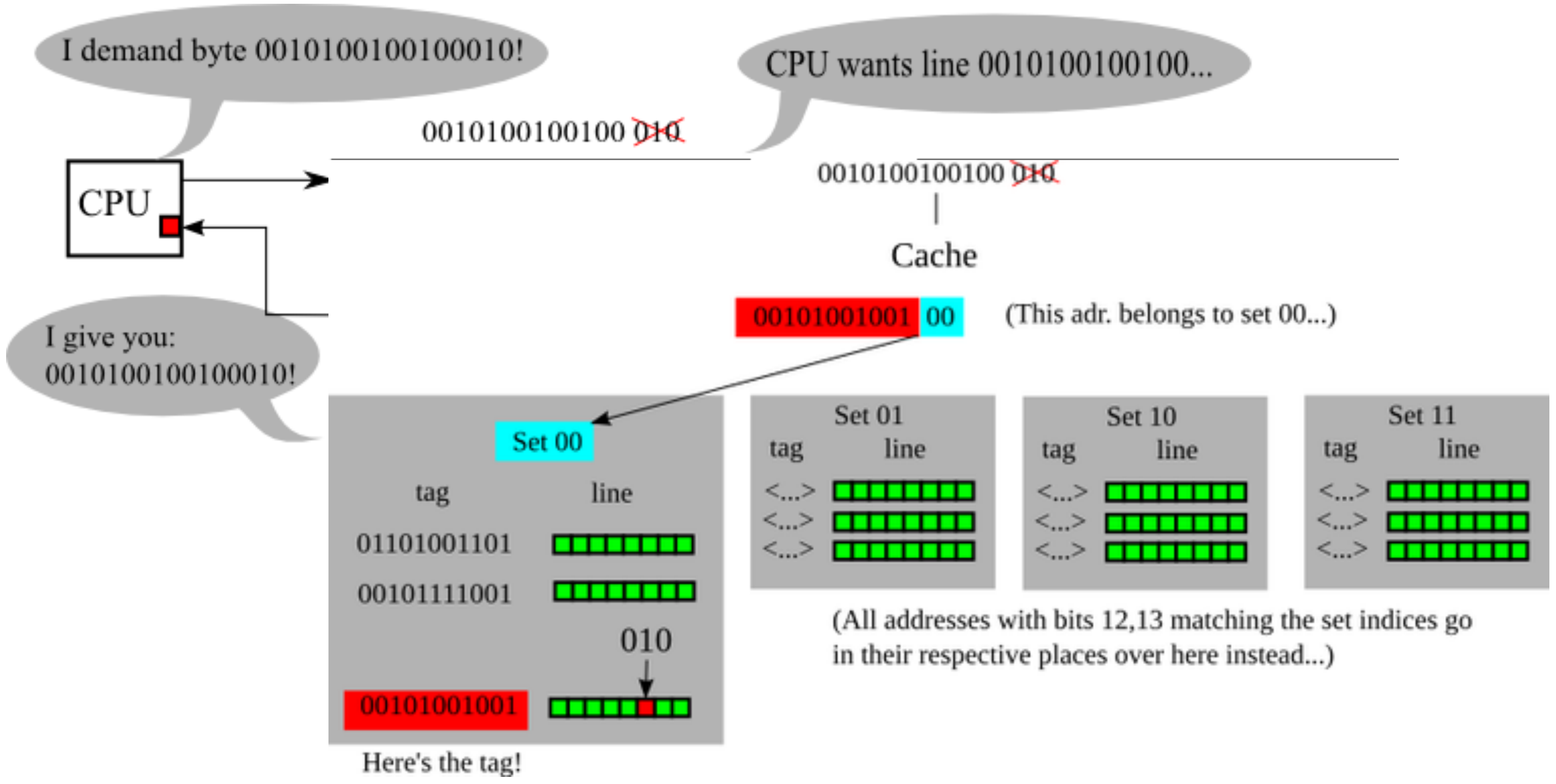
How the cache memory works

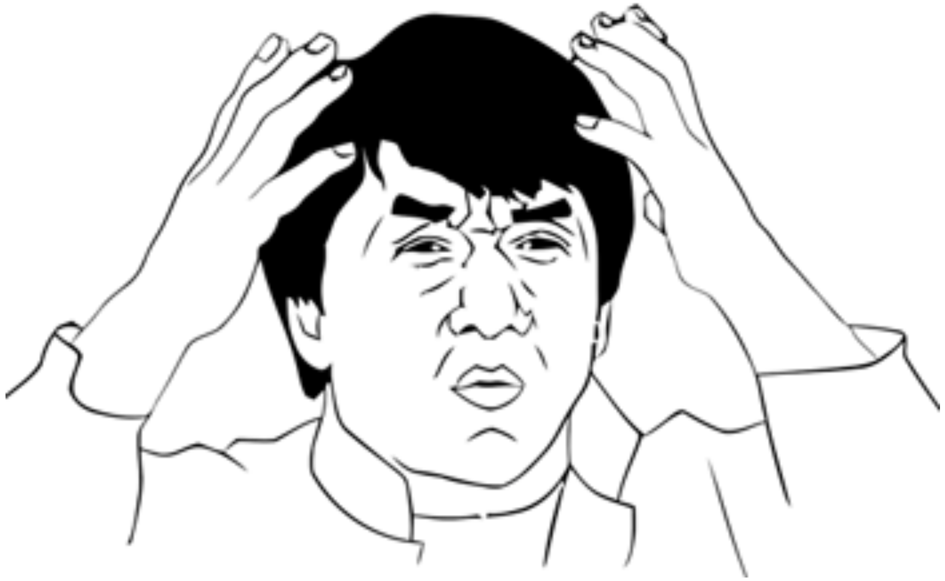
- **Set associative cache(*multi-way-direct-mapped*)**
  - A trade-off for the two previous approaches
  - Any memory block can be stored in any cache way, but only can be stored in one specific cache entry in that way



How the cache memory works

- Cache addressing


I demand byte 0010100100100010!
CPU wants line 0010100100100...
0010100100100 ~~010~~
CPU
I give you: 0010100100100010!
0010100100100 ~~010~~
Cache
00101001001 00
(This adr. belongs to set 00...)
Set 00
tag
line
01101001101
00101111001
010
00101001001
Here's the tag!
Set 01
tag
line
<...>
Set 10
tag
line
<...>
Set 11
tag
line
<...>
(All addresses with bits 12,13 matching the set indices go in their respective places over here instead...)

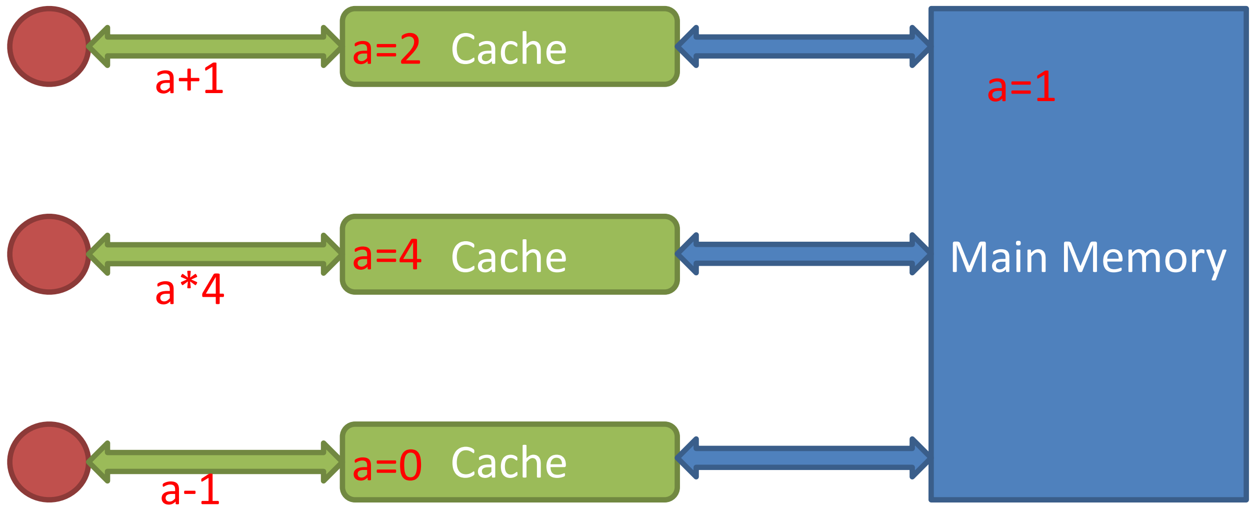


Enough! So what about me?

Let’s look at concurrency programming…

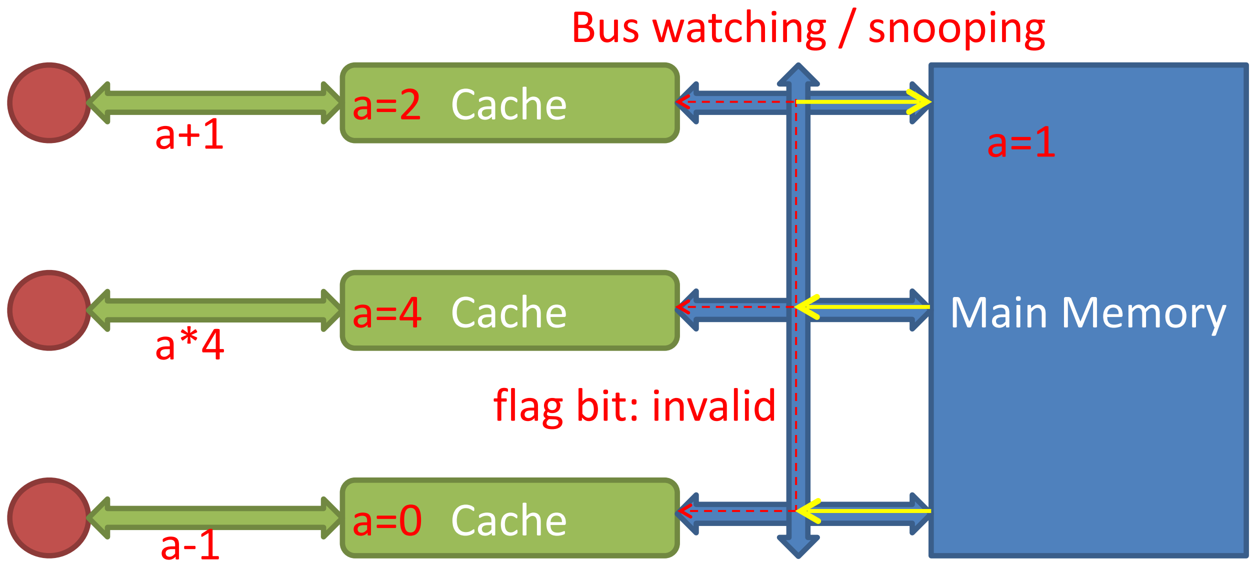
Concurrency programming

- Atomicity
- Visibility of memory
- Shared & mutable variables



Concurrency in Java

- Volatile
  - Lock# instruction(cache line locked)
  - Cache coherency(MESI protocol, modified, exclusive, shared, invalid)
  - Bus watching / snooping(use of a bus "shared line" to detect "shared" copy in the other caches)



Concurrency in Java

- Volatile
- Happens-before
- Memory Barriers(CPU instructions)
- Lock-free
- Pseudo-concurrency

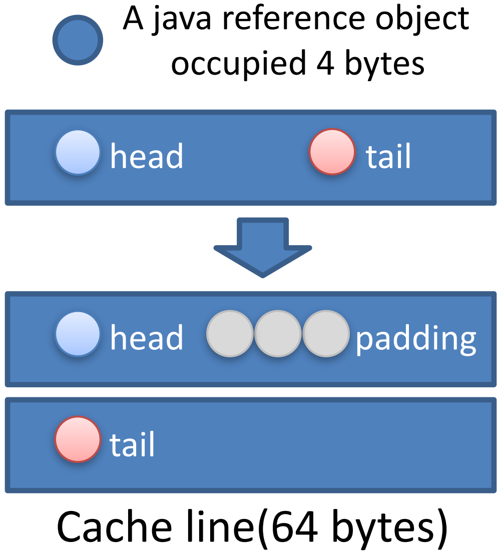
Cache lines (write-buffer) flush back to main memory

```
//Share variables:
int a = 0;
volatile int b = 1;
```

```
//Thead-1:
a = 5;
b = 2;
```

```
//Thead-2:


print a; //5
print b; //2
```



```
/** head of the queue */
private transient final PaddedAtomicReference<QNode> head;

/** tail of the queue */
private transient final PaddedAtomicReference<QNode> tail;

static final class PaddedAtomicReference <T> extends AtomicReference <T> {

  // enough padding for 64bytes with 4byte refs
  Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;

  PaddedAtomicReference(T r) {

    super(r);

  }

}
```

LinkedTransferQueue.java

CPU Cache

- Cache levels
- Cache structure
- Cache addressing
- Cache associativity
- Cache policy(write & replacement)
- Cache hit/miss
- Cache coherency
- Cache hierarchy in CPU

Reference

- https://computing.llnl.gov/tutorials/parallel_comp/
- https://en.wikipedia.org/wiki/CPU_cache
- https://en.wikipedia.org/wiki/Cache_memory
- http://www.hardwaresecrets.com/how-the-cache-memory-works/
- https://www.quora.com/How-does-the-cache-memory-in-a-computer-work
- http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI
- https://superuser.com/questions/196143/where-exactly-l1-l2-and-l3-caches-located-in-computer
- https://en.wikipedia.org/wiki/Disk_sector
- https://en.wikipedia.org/wiki/Cylinder-head-sector
- https://en.wikipedia.org/wiki/Zone_bit_recording
- http://www.tldp.org/LDP/sag/html/hard-disk.html
- https://www.youtube.com/watch?v=Cj8-WNjaGuM&list=PLlVZ1eXuYslrb9G6xm4SKV51SO-KAFrCw&index=1&t=12s
- http://blog.csdn.net/hguisu/article/details/7408047
- http://www.pcguide.com/ref/hdd/geom/tracksZBR-c.html
- http://cn.linux.vbird.org/linux_basic/0230filesystem.php
- https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/10_MassStorage.html
- https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics
- https://en.wikipedia.org/wiki/I/O_scheduling
- https://www.howtogeek.com/115229/htg-explains-why-linux-doesnt-need-defragmenting/
- https://en.wikipedia.org/wiki/Readahead
- https://en.wikipedia.org/wiki/B-tree
- https://en.wikipedia.org/wiki/B%2B_tree
- http://www.cnblogs.com/yangecnu/p/Introduce-B-Tree-and-B-Plus-Tree.html
- http://www.cs.umb.edu/~poneil/lsmtree.pdf
- http://www.cnblogs.com/siegfang/archive/2013/01/12/lsm-tree.html