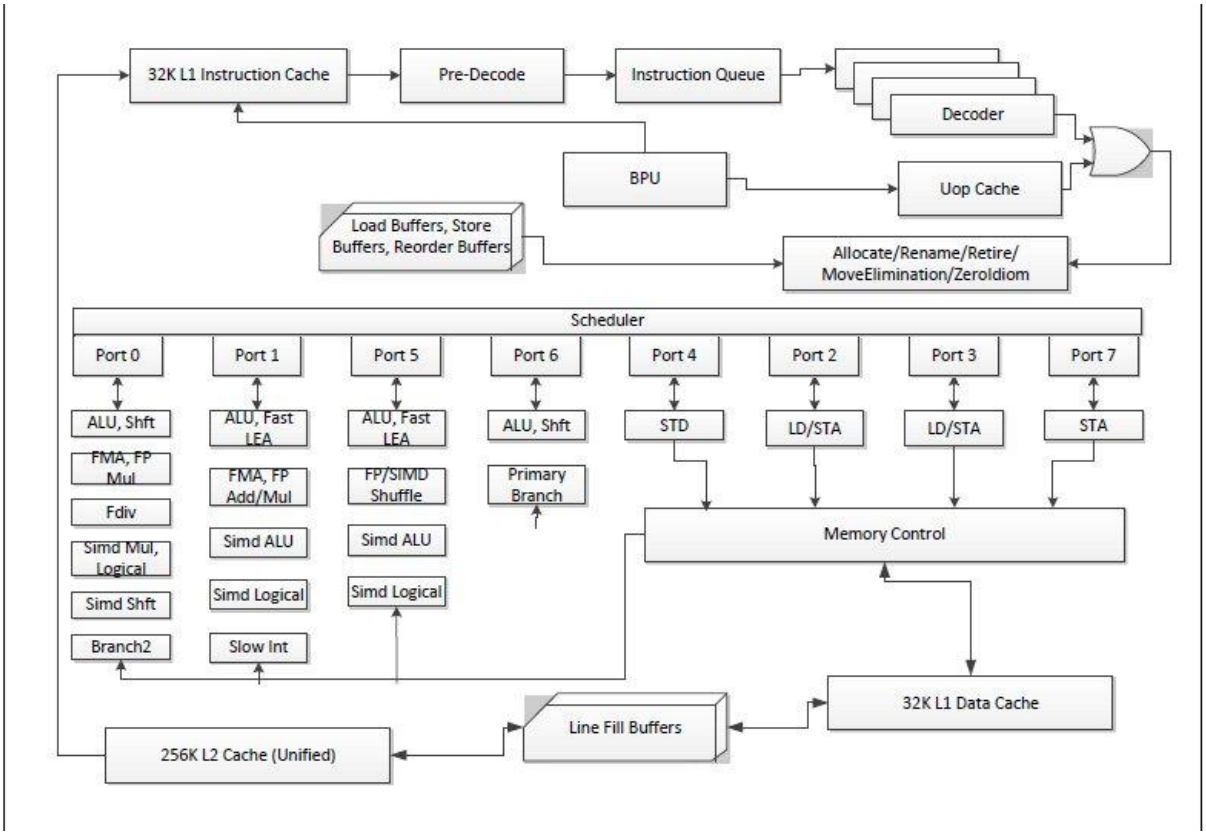

Intel Haswell CPU pipeline

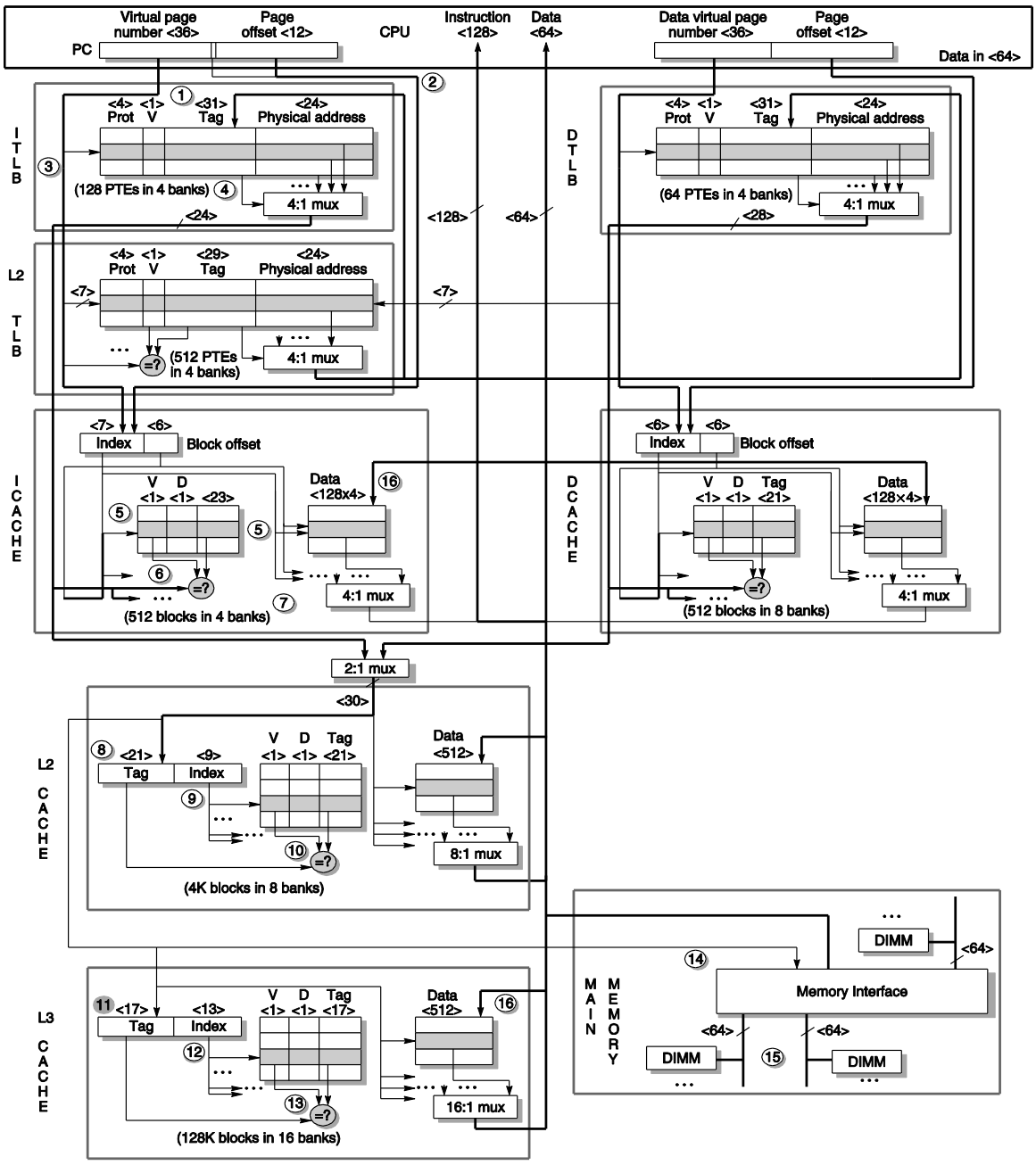

Fig. 2.21 Intel i7 memory architecture (Computer Architecture – A quantitative approach)

Cache Memory

Cache memory is used by the CPU to reduce the average time to access memory.

Cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations.

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory.

Data is transferred between memory and cache in blocks of fixed size, called *cache lines*, which is, e.g., 64 bytes of data

Most modern desktop and server CPUs have at least three independent caches:

- an *instruction cache* to speed up executable instruction fetch,
- a *data cache* to speed up data fetch and store,
- a t*ranslation lookaside buffer* (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data.

The data cache is usually organized as a hierarchy of cache levels. Larger levels are slower.

- 28 μop Decoder Queue (caches decoded instructions for small loops)
- 1.5K μop cache (caches decoded instructions)
- ITLB – 128 entries (virtual memory address translation for instructions)
- DTLB0 – 16 entries, loads only (virtual memory address translation for data)
- DTLB1 – 256 entries, loads and stores (virtual memory address translation for data)
- 32kB L1data cache
- 32KB L1 instruction cache
- 4MB L2 cache (shared by pairs of caches)
- 8MB L3 cache (shared by all the processor cores)
- PDE cache (top level page table access) 64 entries
- PDP cache (next level page table access) 4 entries

Check the smallest level 1 (L1) cache first; if it hits, the processor proceeds at high speed. If the smaller cache misses, the next larger cache (L2) is checked, and so on, before external memory is checked.

Latency (Intel i7-3960X : 3.9GHz, 6 cores & 12 threads, L1:6x 32kB / L2:6x 256kB / L3:15MB):

CPUs *prefetch* data into the caches speculatively, i.e. they guess which instruction/data will be needed next and fetch it to be ready when needed.

Prefetching does not work as well with irregular memory patterns.

A random access pattern causes TLB misses.

Access	**L1 data**	**L2**	**L3**	**memory**
Sequential (no TLB miss):	3 clk	11 clk	14 clk	6 ns
In-page (no TLB miss):	3 clk	11 clk	18 clk	22 ns
Random:	3 clk	11 clk	38 clk	65 ns

The constants 6 and 65 are significantly different, but Big-Oh ignores this difference.

Some of the latency is hidden by the hardware.

For example, the pre-fetcher may fetch data from RAM into L1 that it predicts you will need, so that the data is in L1 by the time you need it.

Hence, the reported 6 ns access time (instead of the actual 150ns time) for sequential accesses. (This sequential pattern is easily identified by the pre-fetcher).

Q: Which loop is faster?

```
int arr[] = new int[16 * 1024 * 1024];

Loop1: For (int i=0; i< arr.length; i++) arr[i] *=3;

Loop 2: For (int i=0; i< arr.length; i+=16) arr[i] *=3;
```

They take about the same time, but second loop does 6% of the work as the first. Why?

A: 16 ints = 64 bytes, size of cache line. Both loops touch the same number of cache lines.

Q: int[][] = 1 2 3 which elements are next to each other? 1-2 or 1-4
4 5 6