A little about hardware

Table of contents

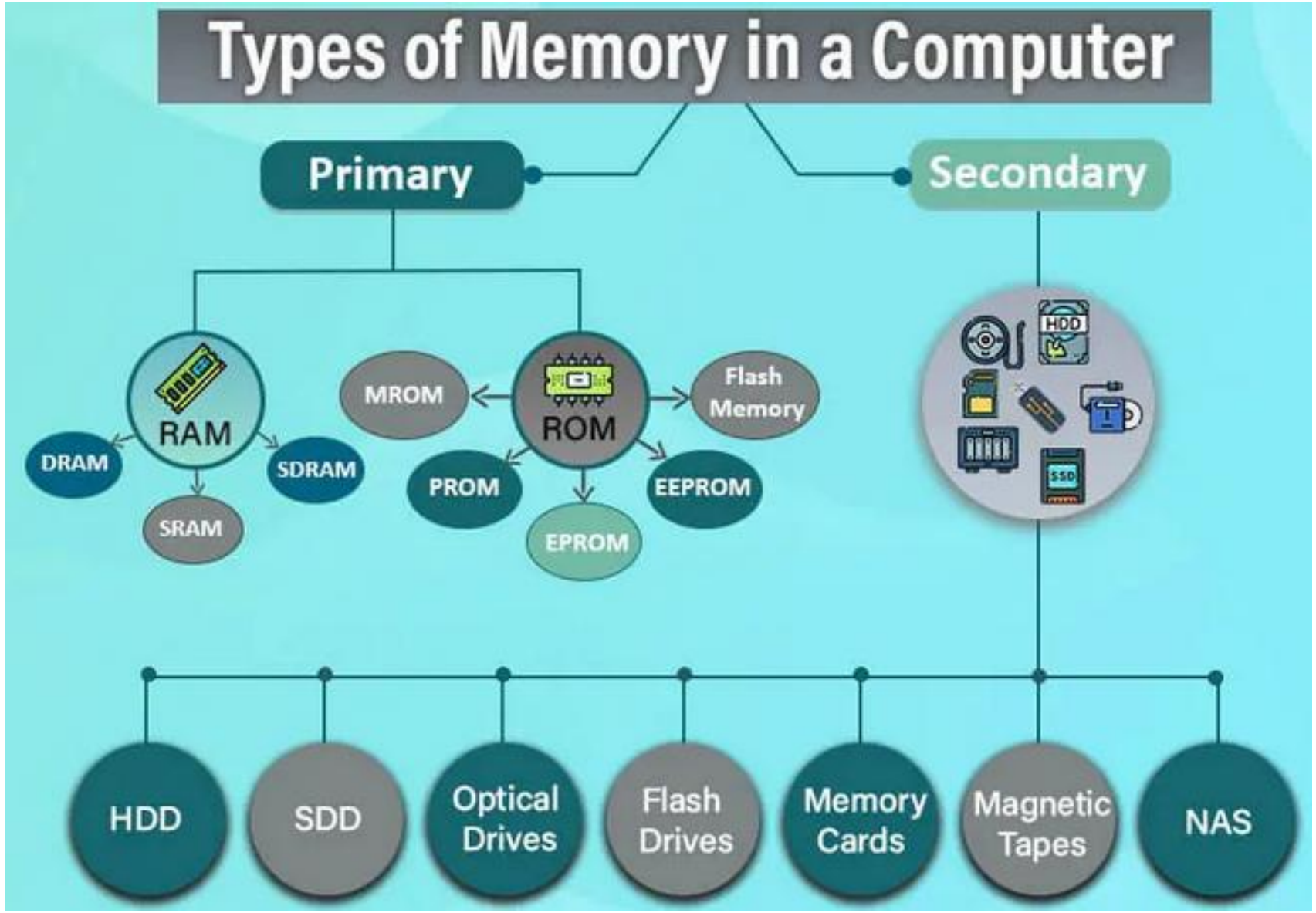
Processor architectures

- **Processor architectures** describe the design and organization of a computer's central processing unit (**CPU**). They determine the data path width, integer size, and memory address width that a processor can handle.
- There are three types of microprocessors:
  - **Complex Instruction Set Computer** (**CISC**) is a processor known for its complex instruction set and high code density. CISC processors have a rich set of instructions that can perform a variety of tasks, which results in high code density and efficient use of memory.

    Examples: **x86 – Intel 386**, **Intel 486**, **Pentium**, **Pentium Pro**, **Pentium II A**
  - **Reduced Instruction Set Computer** (**RISC**) is a processor characterized by its simple instruction set compared to CISC processors. The result is faster execution and less power consumption. This architecture is known for its high performance and energy efficiency, making it appropriate for computing applications such as servers, supercomputers, and embedded systems.

    Examples: **ARM, PowerPC, RISC-V, SPARC**
  - **Explicitly Parallel Instruction Computing** (**EPIC**) allows the instructions to compute in parallel by making use of compilers. This was intended to allow simple performance scaling without resorting to higher clock frequences. (Not to confused with EPYC which is a brand of multi-core x86-64 microprocessors designed and sold by AMD.)

    Examples: **IA-64** (**Intel Architecture-64**) – **Itanium**, **Itanium 2**
- There are two primary word sizes used in today's processor architectures: **32-bit** (x86) and **64-bit** (x86-64, IA64, and AMD64).



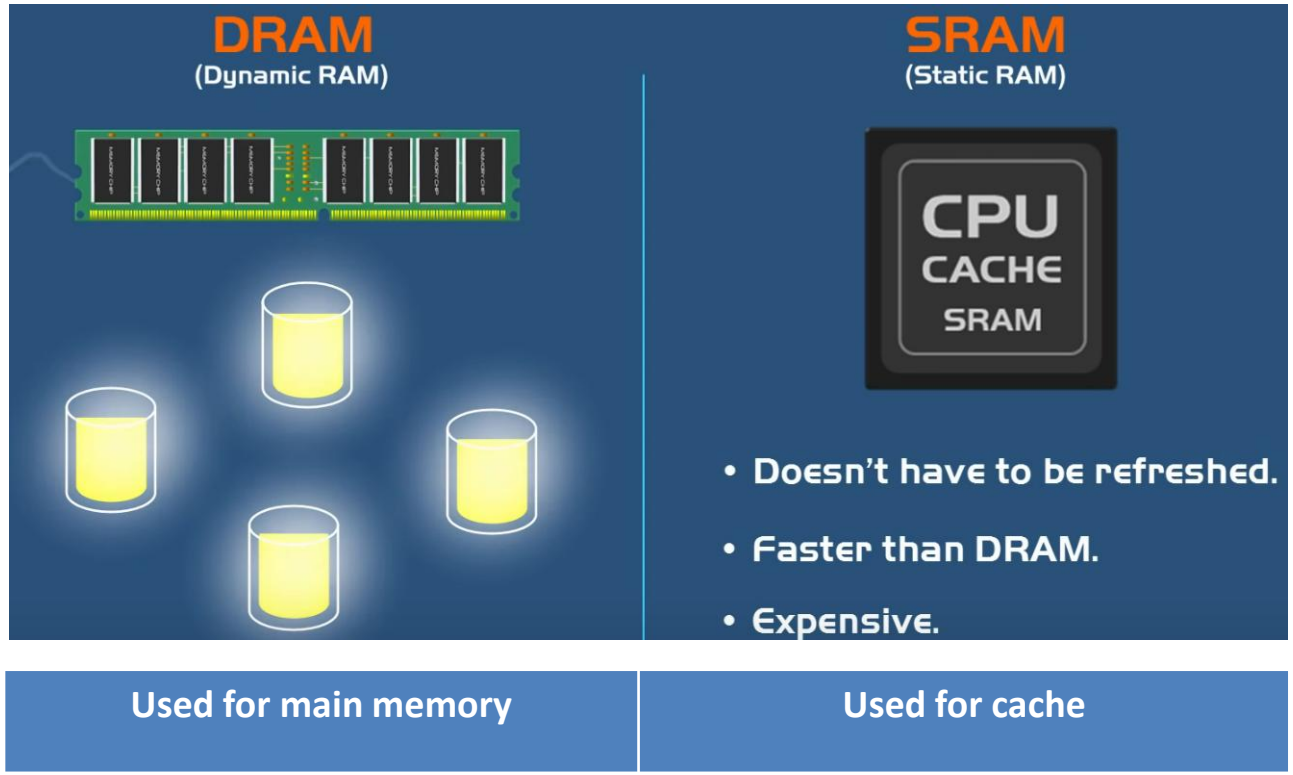
[Types of Memory in a Computer]

Types of memory
Dynamic Random Access Memory

Dynamic Random Access Memory, DRAM – a kind of **volatile** semiconductor memory with **random access**, the bits of which are represented by the state of charge of the capacitors. They require periodic refreshing of the content (use less energy), unlike **static memories**, which require **constant power supply** (use more energy).

Synchronous Dynamic Random Access Memory, SDRAM – the type of DRAM working **synchronously** with the system bus (which distinguishes it from the classic DRAM that works **asynchronously**).

SDRAM family includes:
SDR (*Single Data Rate*),
DDR (*Double Data Rate*), DDR2, DDR3, DDR4, DDR5.



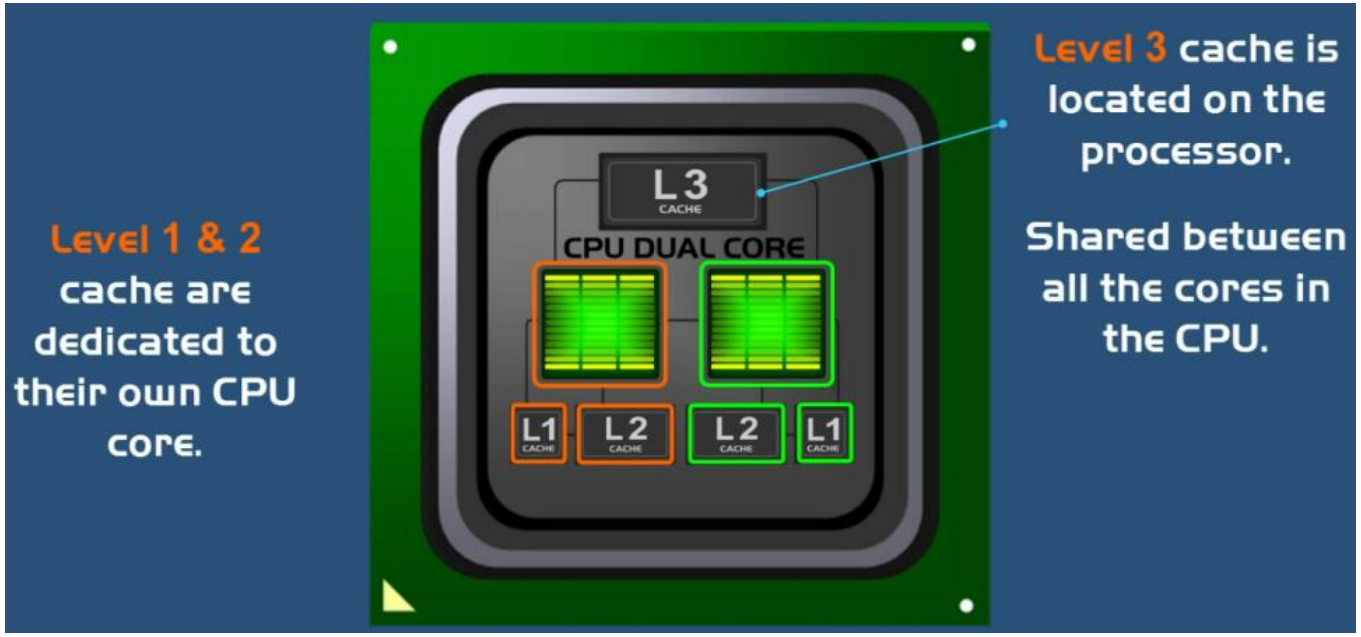
https://www.youtube.com/watch?v=yi0FhRqDJfo

Types of memory
Static Random Access Memory

SRAM memories are used in **fast cache memories**,

- they do not require **large capacities** (data density in SRAM is 4 times lower than in DRAM),
- access speed is about **7 times faster** than DRAM (**1 SRAM cycle is about 10 ns**, while in **DRAM about 70 ns**).

This speed applies to random access, in the case of reading data from neighboring address cells, the speed of SRAM and DRAM is comparable.



Cache is crucial because it eliminates the differences in **RAM** and **CPU speeds**.

https://www.youtube.com/watch?v=yi0FhRqDJfo

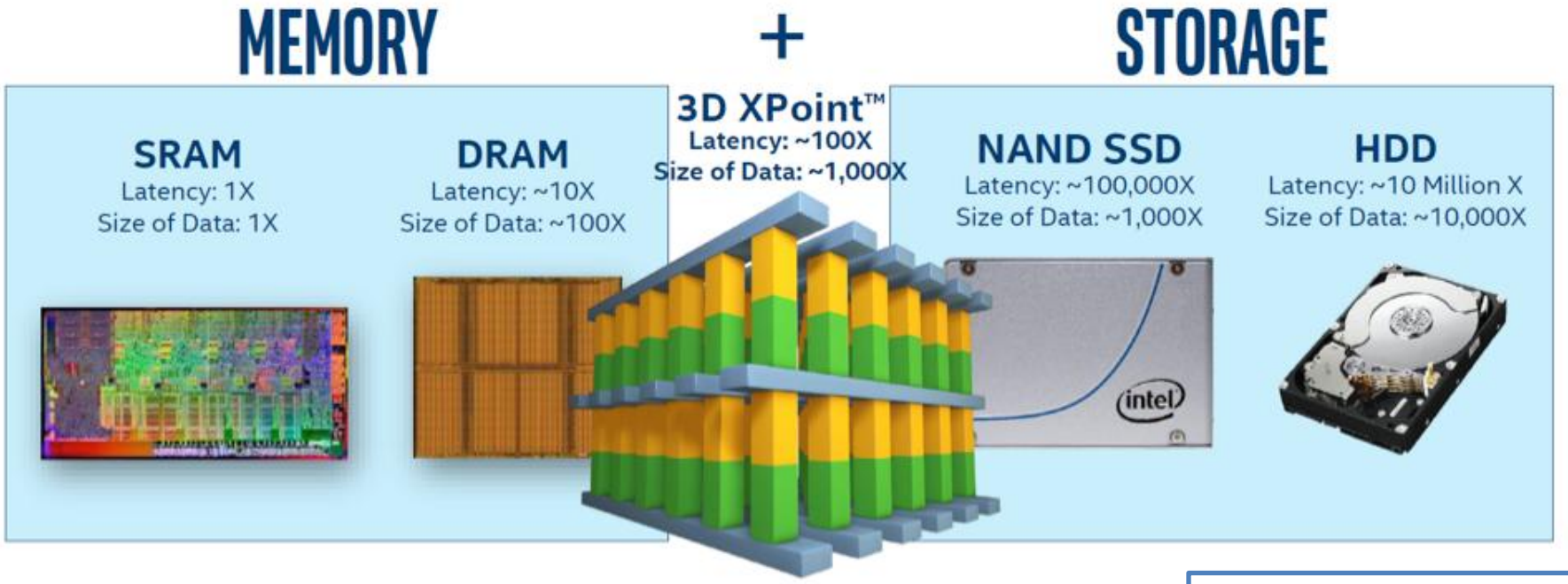


Image source: Intel

A [non-volatile memory](#) (NVM) technology.

In 2015, **Intel** and **Micron** claimed **3D XPoint** would be up to **1,000 times faster** and have up to **1,000 times more endurance** than NAND flash, and have **10 times the storage density** of conventional memory. Up to **half the cost** of DRAM.

2023 – discontinued

Available on the open market under the brand name **Optane** (Intel) since April 2017.

[Intel 3D XPoint Technology](#), Disruptive Technologies Session, 2015 HPC User Forum

[3D XPoint™ Technology Revolutionizes Storage Memory](#)

Admiral Grace Hopper explains the nanosecond

6,2 tys. km

1906-1992

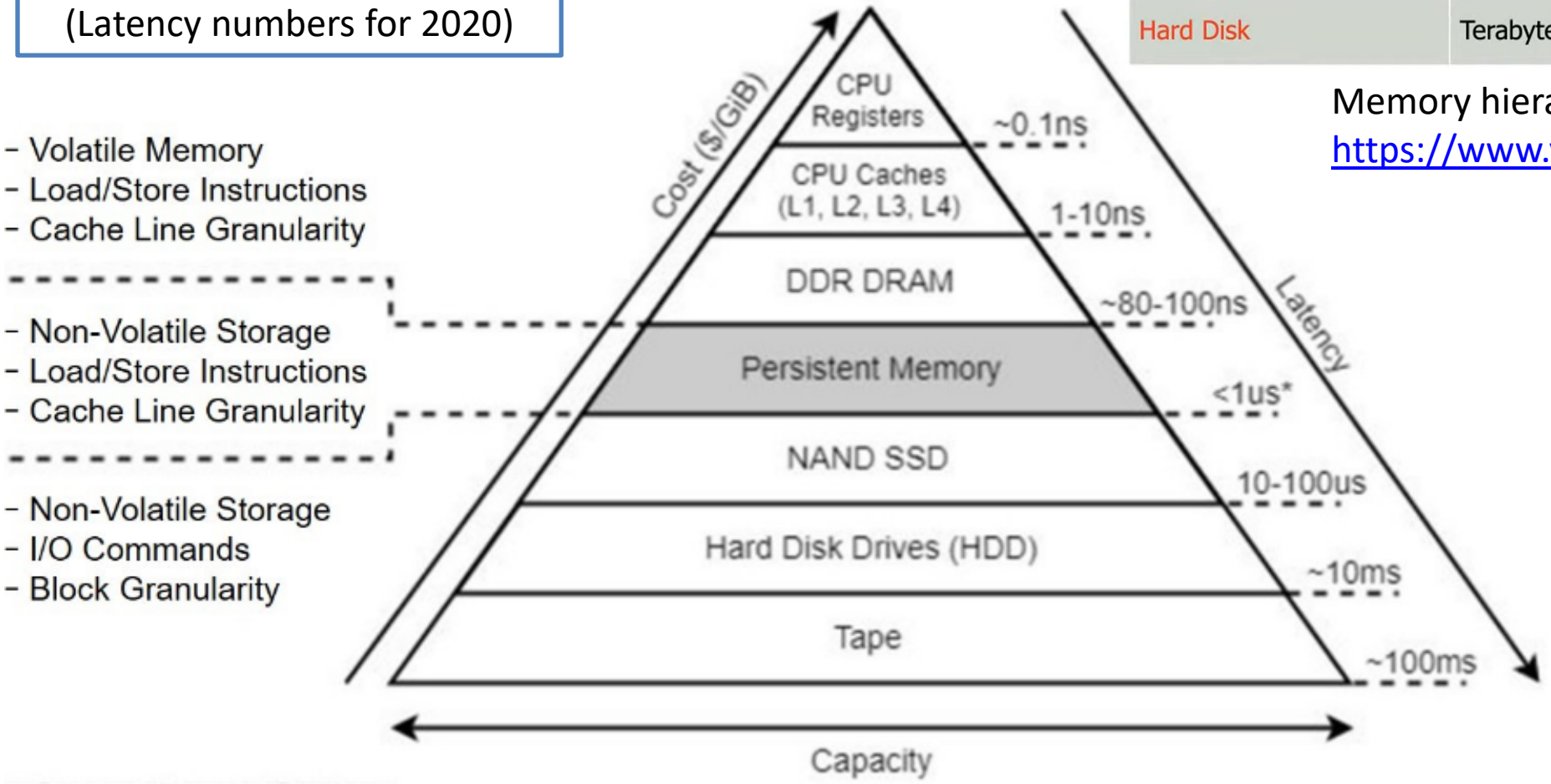
- I called over to the engineering building and I said: „Please cut off a nanosecond and send it over to me”.
- I wanted a piece of wire which would represent the maximum distance that electricity could travel in a billionth of a second. Of course, it wouldn’t really be through wire. It’d out in space; **the velocity of light**.
- So, if you start with the velocity of light, you’ll discover that a **nanosecond** is **11.8 inches** long (**29,97 cm**)

- At the end of about a week, I called back and said: „I need something to compare this to. Could I please have a microsecond?”
- Here is a **microsecond**, **984 feet** (**29992,32 cm**). I sometimes think we ought to hang one over every programmer’s desk (or around their neck) so they know when they’re throwing away when they throw away microseconds.

https://www.youtube.com/watch?v=9eyFDBPk4Yw

Memory Device	Capacity	Latency	Cost per Megabyte
SRAM	512 Bytes	sub-nanosec	
SRAM	KByte~MByte	~nanosec	< 0.3$
DRAM	Gigabyte	~50 nanosec	< 0.03$
PCM-DIMM (Intel Optane DC DIMM)	Gigabyte	~200 nanosec	< 0.004$
PCM-SSD (Intel Optane SSD)	Gigabyte ~Terabyte	~10 µs	< 0.001$
Flash memory	Gigabyte ~Terabyte	~100 µs	< 0.00008$
Hard Disk	Terabyte	~10 millisec	< 0.00003$

[Memory hierarchy, Intel](#)
(Latency numbers for 2020)



Memory hierarchy, sample values ~2021
https://www.youtube.com/watch?v=J6jkrDlgflo

At what speeds different parts of the computer work?

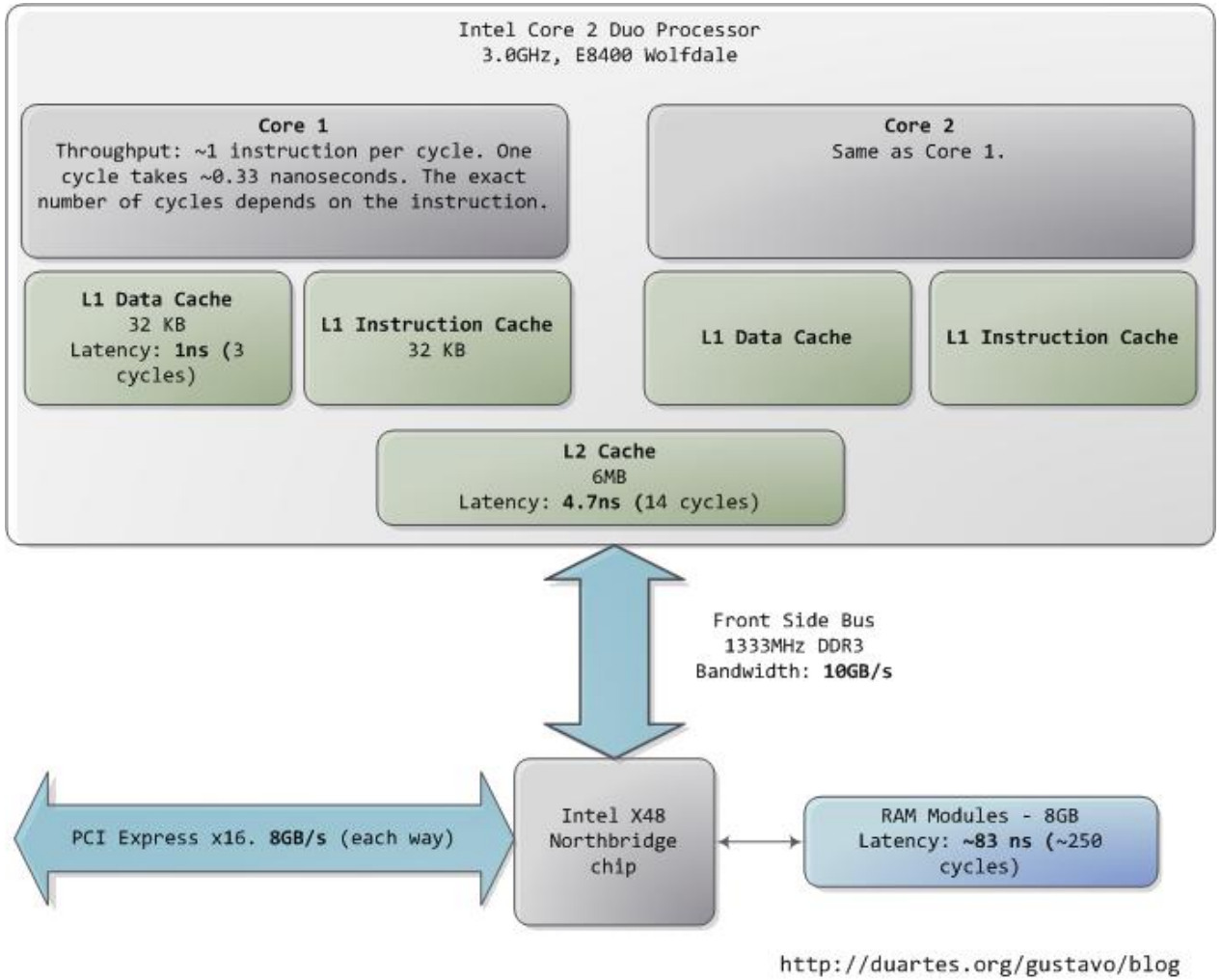
What are
- speed – latency,
- throughput

of various subsystems in a commodity PC (**Intel Core 2 Duo at 3.0 GHz**).

Time units are:
- nanoseconds (ns, 10^{-9} s),
- milliseconds (ms, 10^{-3} s),
- seconds (s).

Throughput units are in **megabytes** and **gigabytes** per **second**.

https://en.wikipedia.org/wiki/X86



Processor and northbridge chip
(source: Duarte, Software Illustrated)

Hard disks also have **caches**.

They are **small** in size (16 MB cache is only 0.002% 750 GB of disk), but thanks to them the **disk can queue up writes** and then perform them in one bunch.

Reads can also be **grouped** in this way for performance, and both the OS and the drive firmware engage in these optimizations.

Standard **SSD** can read sequential data at a **speed** of about 550 megabytes per second (MBps) and write it at 520 MBps. A fast **HDD** may carry out sequential reads and writes at just 125 MBps.

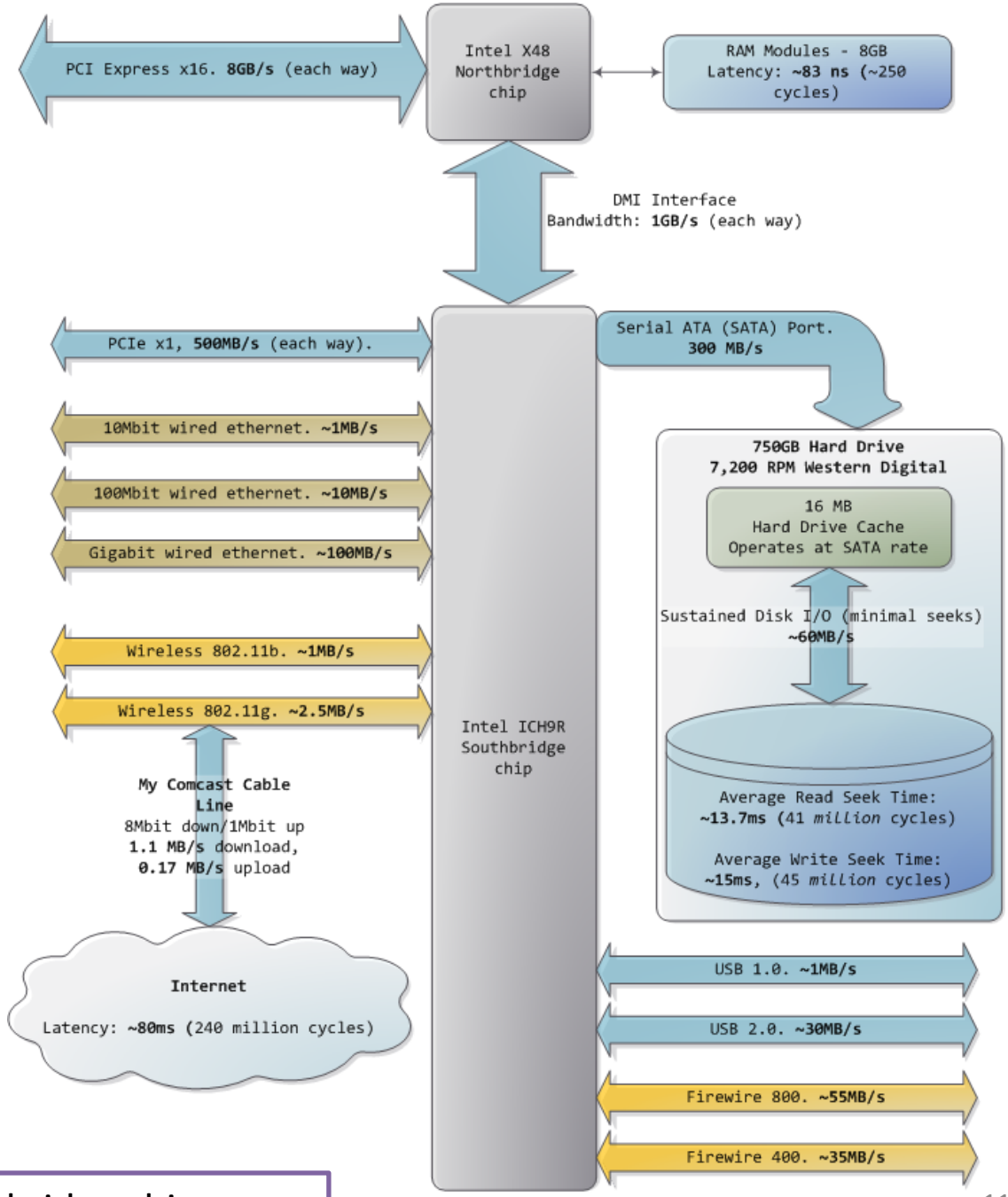
USB 3.0 can transfer data at up to **5 Gbit/s** (625 MB/s), which is about **10 times faster** than the **USB 2.0** standard.

Wired Ethernet – 2.5 Gbit/s (soon 10 Gbit/s)

Wireless 802.11ac – 200 Mbit/s

The **latency** to a **fast website** is about **45 ms**, comparable to **hard drive seek latency**.

The vision of "the **Internet as a computer**" is real!



Southbridge chip

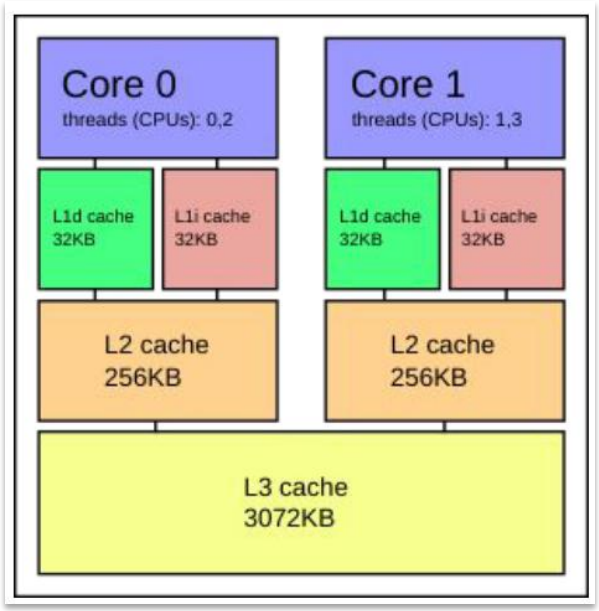
But ...

Brendan Gregg

Be aware, shouting in the datacenter
is not recommended

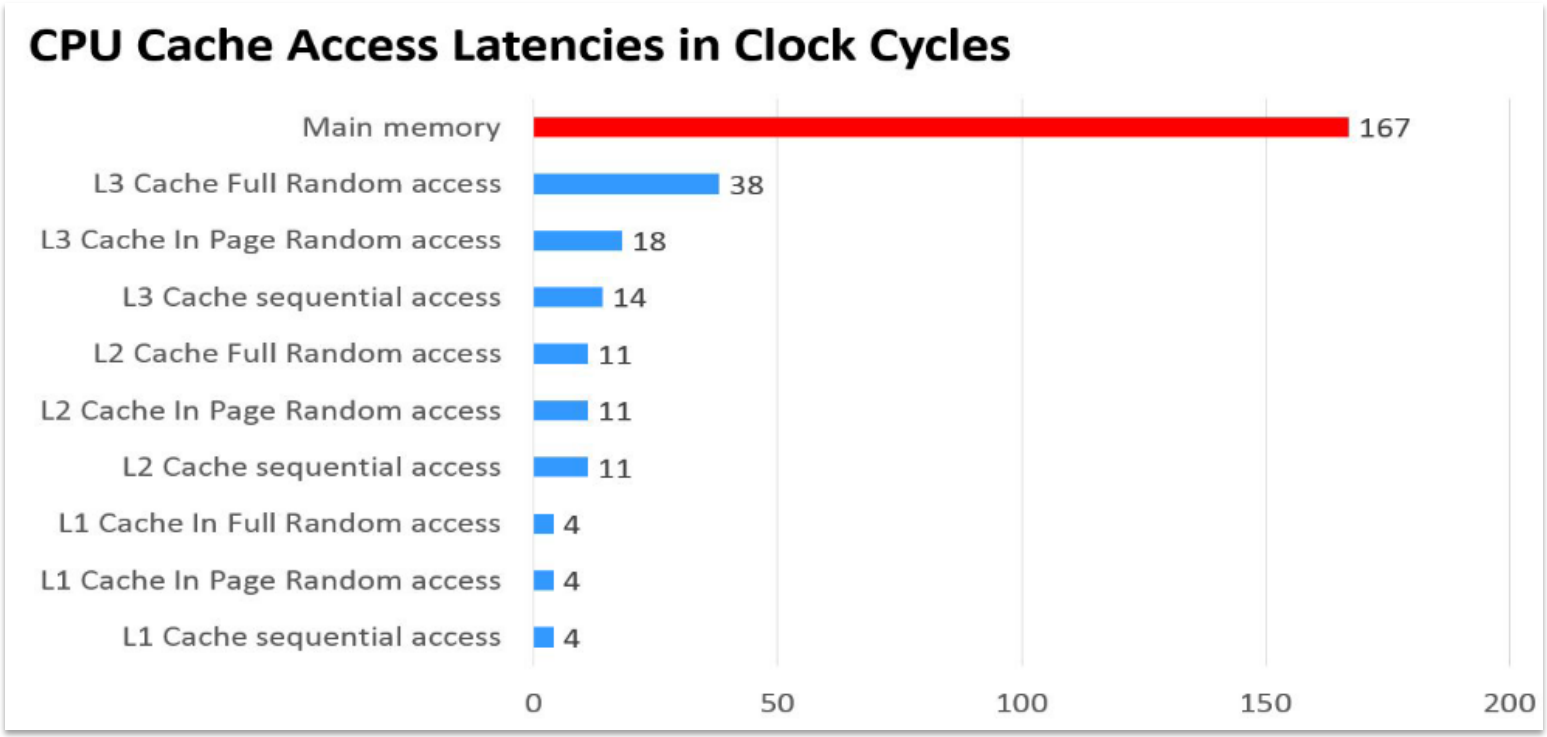
Vibration can badly influence disk latency

https://www.youtube.com/watch?v=tDacjrSCeq4
https://www.youtube.com/watch?v=lMPozJFC8g0



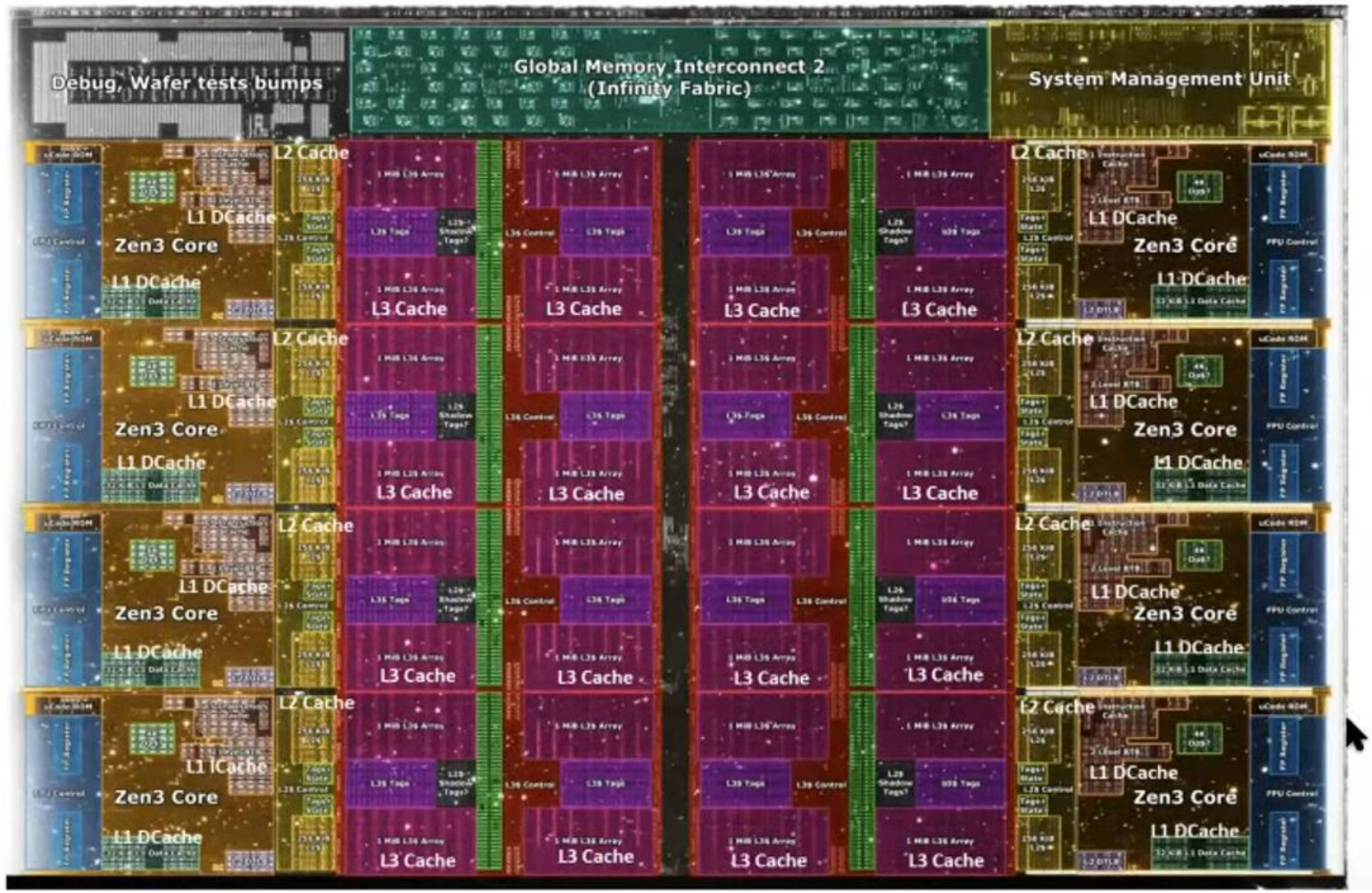
Why do we need the processor cache?

Modern processors use **cache** to store recently used memory cells. It allows to overcome differences in processor speed and memory access time (the **processor** is **much faster**, in addition there may be **several** of them **competing** for **access** to the **same memory**).



Flavors of Memory supported by Linux, their use and benefit, Christopher Lameter (source: presentation at Open Source Summit, 2018)

AMD Ryzen 5000, 2020



Core Count:
8 cores/16 threads

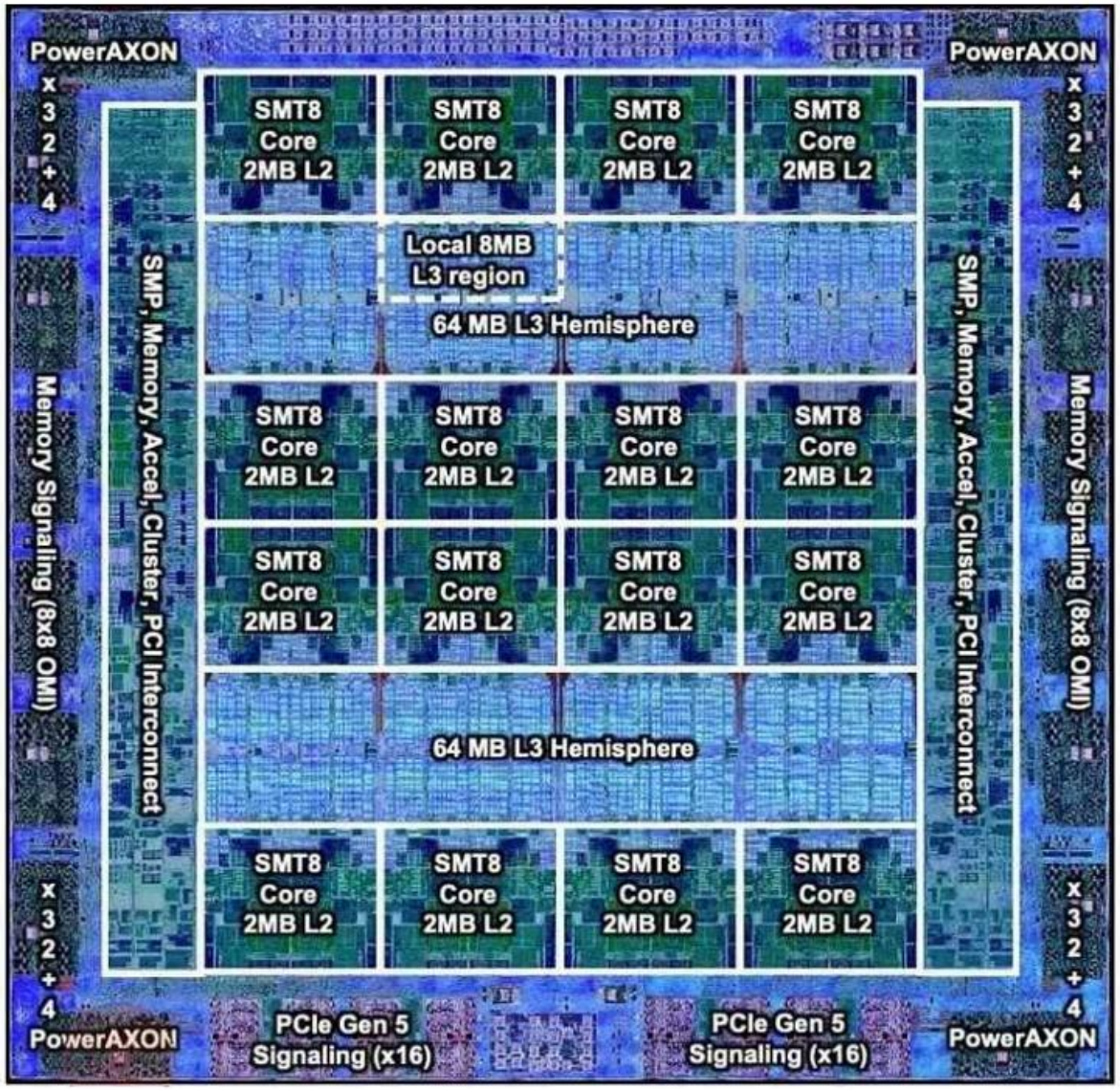
L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

https://www.youtube.com/watch?v=J6jkrDlgflo

IBM POWER10, 2020



Cores:

15-16 cores,
8 threads/core

L2 Caches:

2 MB per core

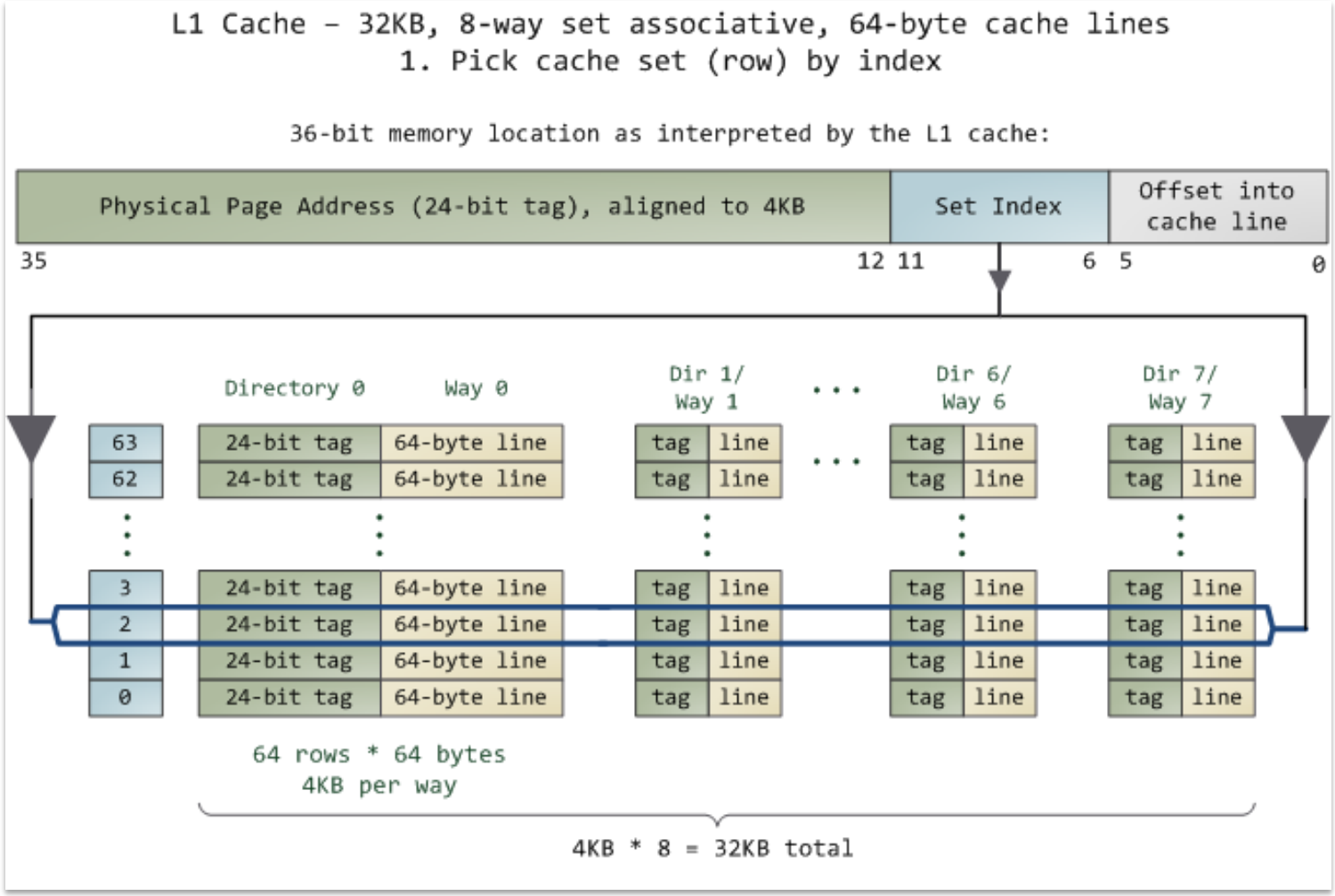
L3 Cache:

120 MB shared

https://www.youtube.com/watch?v=J6jkrDlgflo

How does the processor cache works?

Intel



Processor cache L1 of the Core 2 processor
(source: Duarte, Software Illustrated)

The unit of data in the cache is the **line** (**block**), which is a contiguous chunk of bytes in memory. This cache uses **64-byte lines**.

The lines are stored in **cache banks** or **cache ways**, and each bank has a dedicated **directory** to store its metadata.

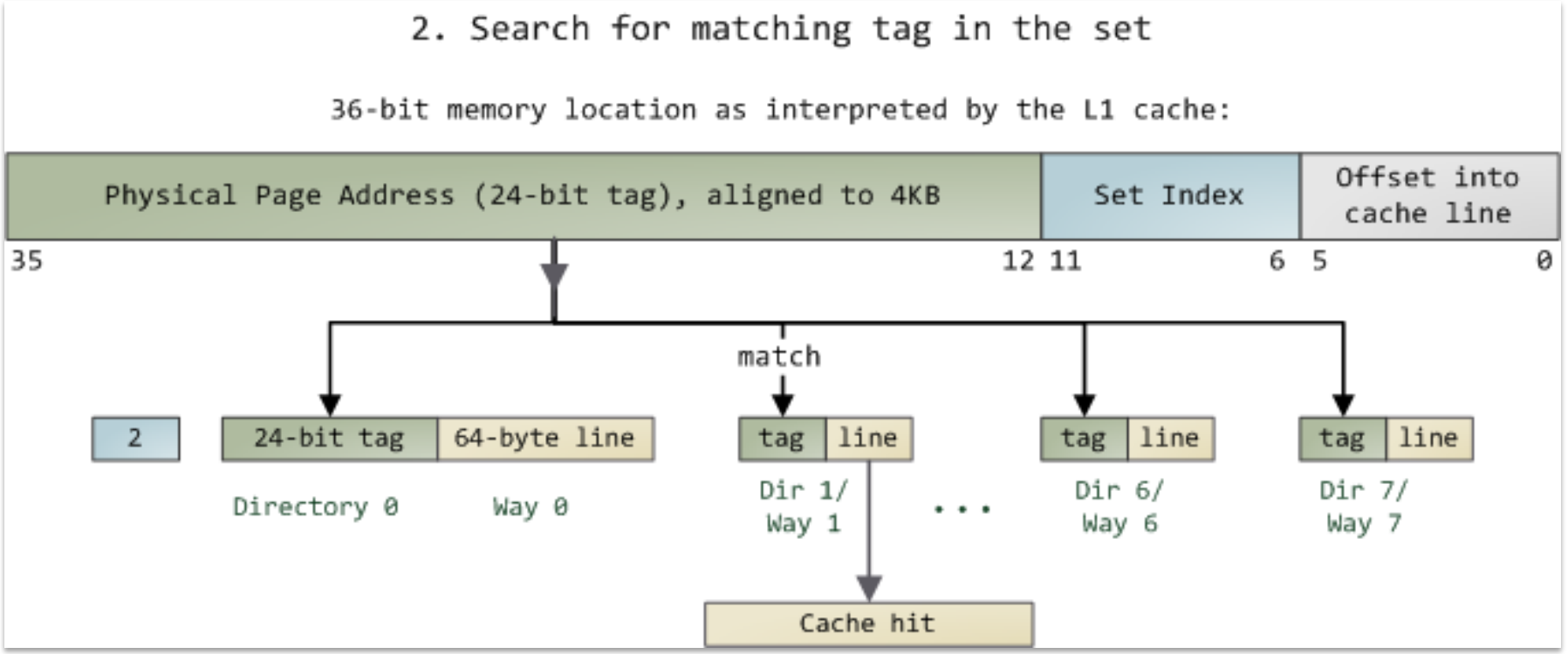
This particular cache has **64 sets** and **8 ways**, hence **512 cells** to store cache lines, which adds up to **32 KB** of space.

How does the processor cache works?

Each **cached line** is **tagged** by its corresponding **directory cell**; the tag is the **number** for the **page** where the line came from.

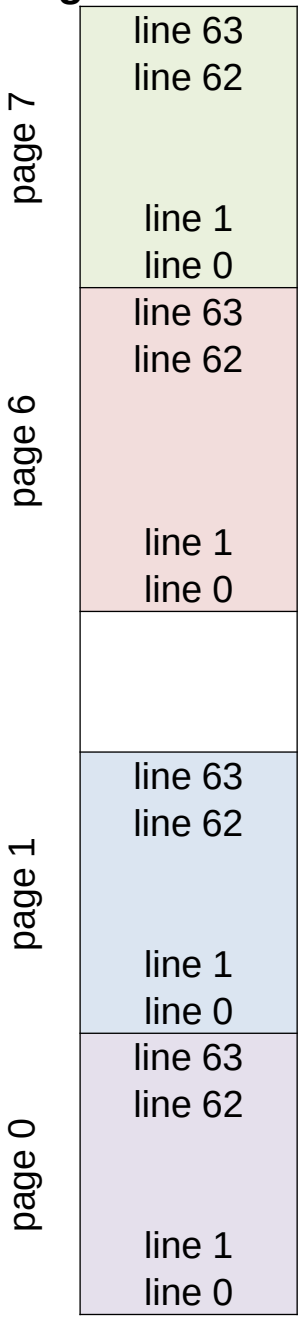
The processor can address 64 GB of physical RAM, so there are 64 GB/4 KB = 2^{24} of these pages and thus we need **24 bits** for the **tag**.

Intel



Search for matching tag in the set (source: Duarte, Software Illustrated)

Pages in memory



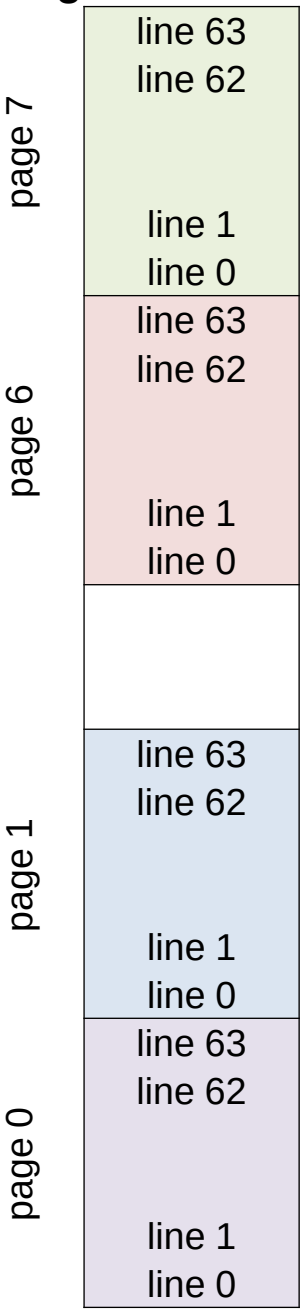
Lines in cache

Dir 0	Way 0	Dir 1	Way 1		Dir 6	Way 6	Dir 7	Way 7	Index
page 0	line 63	page 1	line 63		page 6	line 63	page 7	line 63	63
page 0	line 62	page 1	line 62		page 6	line 62	page 7	line 62	62
page 0	line 1	page 1	line 1		page 6	line 1	page 7	line 1	1
page 0	line 0	page 1	line 0		page 6	line 0	page 7	line 0	0

You can imagine **each bank** and its **directory** as **columns** in a spreadsheet, in which case the **rows** are the **sets**. Each **cell** in the **way column** contains a **cache line**.

Physical memory is divided into **4 KB physical pages**. Each page has 4 KB/64 bytes = 64 cache lines in it. Bytes 0 through 63 within that page are in the first cache line, bytes 64-127 in the second cache line, and so on. The pattern repeats for each page.

Pages in memory



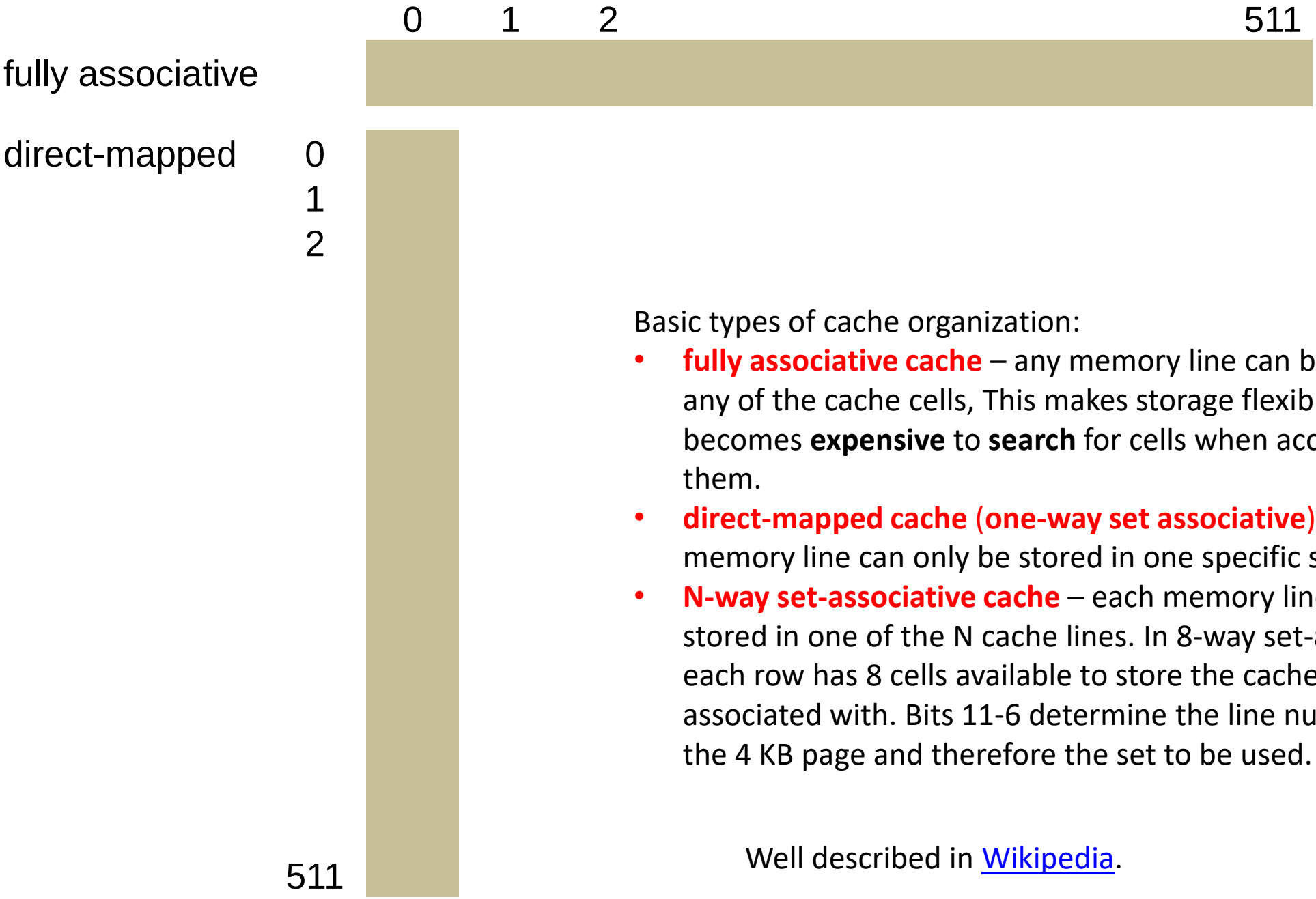
Lines in cache

What happens when we reference

page 100	line 6
page 250	line 4
page 17	line 0
page 342	line 3

(all white slots are occupied)

Dir 0	Way 0	Dir 1	Way 1		Dir 6	Way 6	Dir 7	Way 7	Index
page 1	line 63	page 6	line 63		page 0	line 63	page 7	line 63	63
page 6	line 62	page 1	line 62		page 7	line 62	page 0	line 62	62
page 6	line 1	page 1	line 1		page 7	line 1	page 0	line 1	1
page 0	line 0	page 7	line 0		page 1	line 0	page 6	line 0	0



Basic types of cache organization:

- **fully associative cache** – any memory line can be stored in any of the cache cells, This makes storage flexible, but it becomes **expensive** to **search** for cells when accessing them.
- **direct-mapped cache (one-way set associative)** – a given memory line can only be stored in one specific set (or row),
- **N-way set-associative cache** – each memory line may be stored in one of the N cache lines. In 8-way set-associative each row has 8 cells available to store the cache lines it is associated with. Bits 11-6 determine the line number within the 4 KB page and therefore the set to be used.

Well described in Wikipedia.

How does the processor cache works?

The **tag matching** is **very fast**; electrically all tags are compared simultaneously.

If a set **fills up**, then a **cache line** must be **evicted** before another one can be stored.

To avoid this, performance-sensitive programs try to organize their data so that memory accesses are evenly spread among cache lines.

A memory access usually starts with a **linear (virtual) address**, so the **L1 cache** relies on the **paging unit** to obtain the **physical page address** used for the **cache tags**.

By contrast, the **set index** comes from the least significant bits of the **linear address** and is used **without translation**.

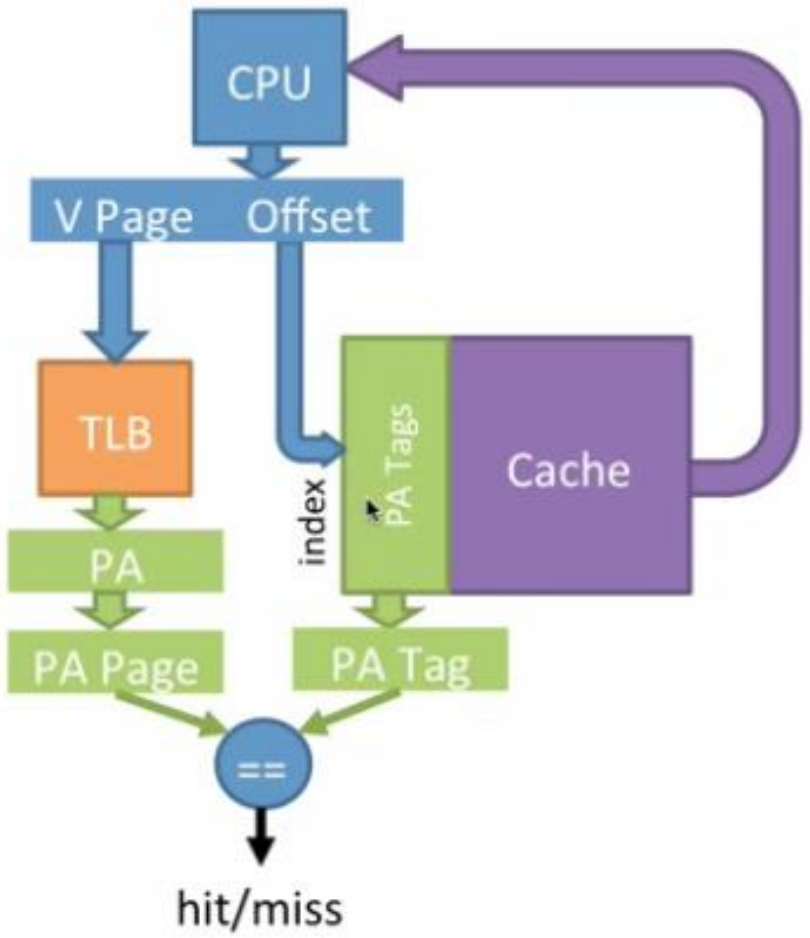
Hence the L1 cache is **physically tagged**, but **virtually indexed**, helping the CPU to parallelize lookup operations.

Because the **L1 bank** is **never bigger** than an **MMU page**, a given physical memory location is guaranteed to be associated with the same set even with virtual indexing.

L2 caches must be **physically tagged and physically indexed** because their **bank size can** be **bigger** than **MMU pages**.

Intel caches are **inclusive**: the contents of the L1 cache are duplicated in the L2 cache.

VIPT: Virtually Indexed, Physically Tagged



TLB translation and **Cache lookup** at the **same time**

- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as the cache

Safe: Cache hit only on PA match

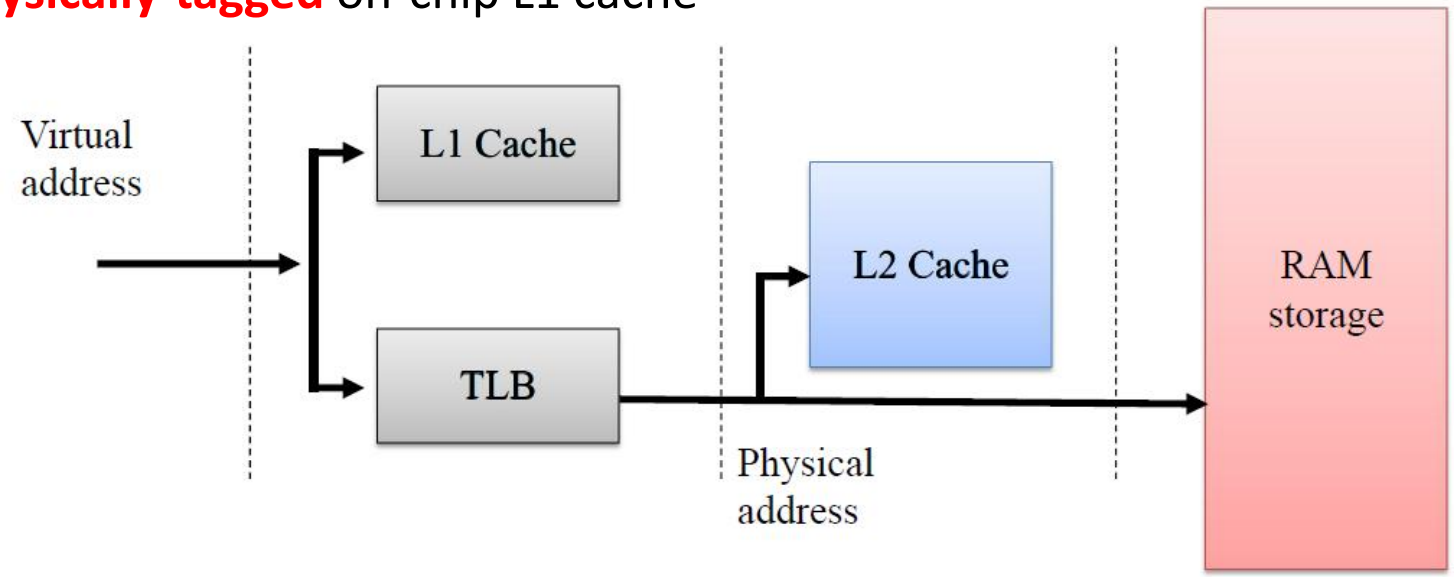
- **But**, can only use non-translated bits to index cache (limit on how large the cache can be)
- Most processors use VIPT for L1 caches today

TLBs and Caches, David Black-Schaffer

Virtual vs physical memory addressing

- **X86** – caches are **physically indexed** and **physically tagged** (except for small L1 caches), virtual address associated with the memory map is filtered by the MMU before real access to the memory hierachy is performed.
- **ARMv4 and ARMv5** – cache is organized as **virtually indexed**, **virtually tagged**, cache lookups are faster (TLB is not involved in matching cache lines for a virtual address), the same physical address can be mapped to multiple virtual addresses.
- **MIPS R4x00** – **virtually-indexed**, **physically-tagged** on-chip L1 cache

In typical architectures, the optimal performance is achieved by having the L1 cache and the TLB **racing** to provide their outputs for subsequent use



A Primer on Modern Hardware Architectures, Alessandro Pellegrini

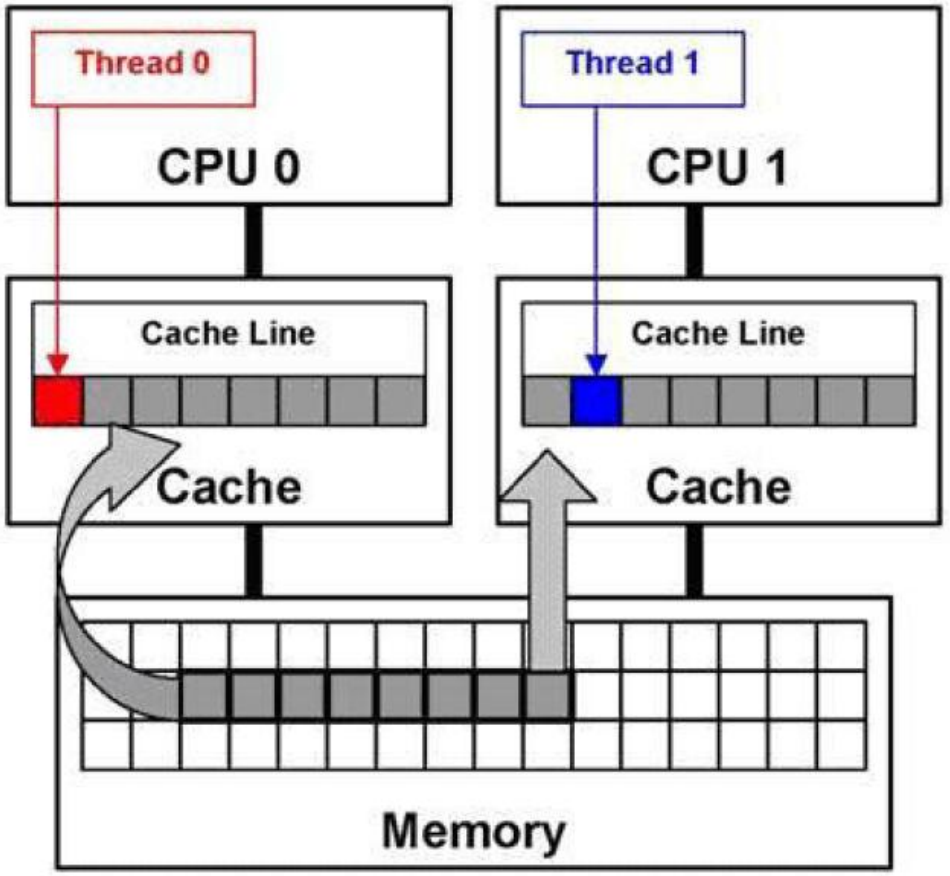
Cache coherence

In multiprocessor systems, each processor has its own cache memory. Additional hardware is needed to synchronize the contents of these memories (**snooping cache**).

Directory cell also stores the state of its corresponding cached line. A line in the L1 code cache is either **Invalid** or **Shared** (which means valid).

There is a whole family of cache coherence protocols (MSI, MESI, MOESI).

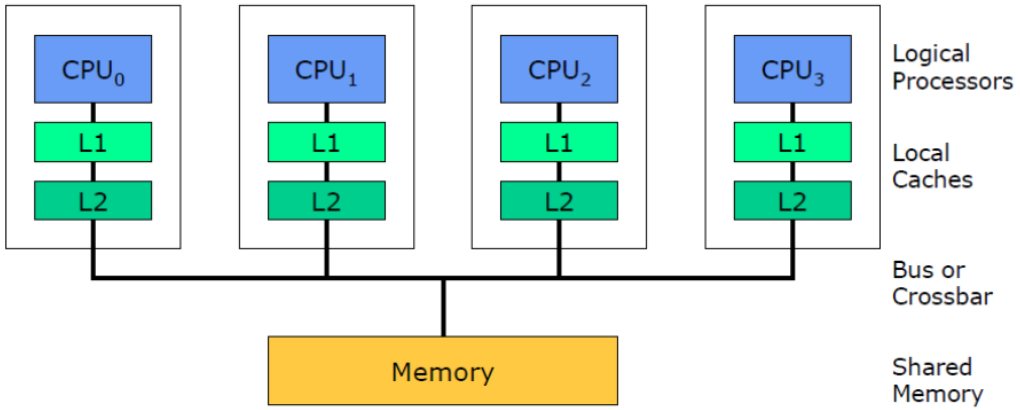
Extra reading: Memory part 2: CPU Caches (Ulrich Drepper, October 2007); Hardware insights, Francesco Quaglia.

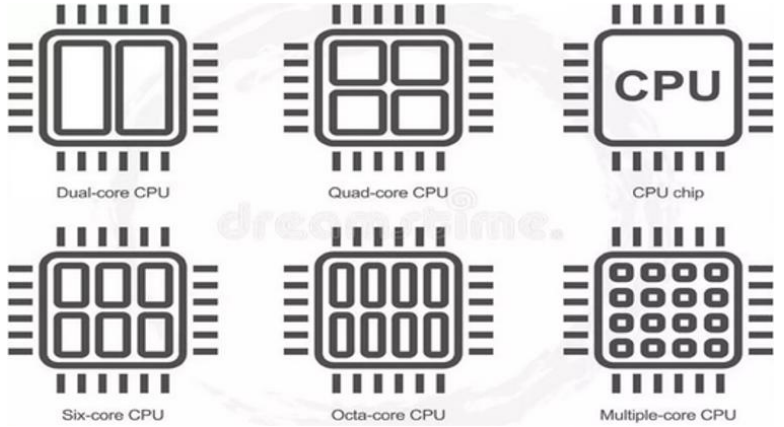


The false **cache sharing** problem

Architectures

Symmetric multiprocessors

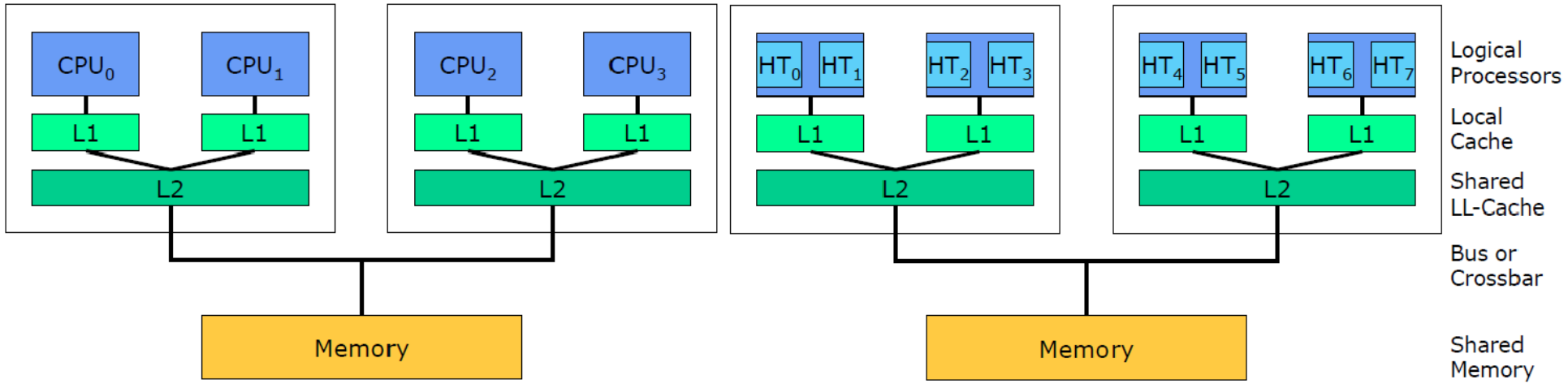




Hyper-Threading Technology enables a single physical processor to execute two or more threads concurrently using shared execution resources. The OS will recognize each physical core as 2 virtual cores.

Chip Multi Processor (CMP) – Multicore

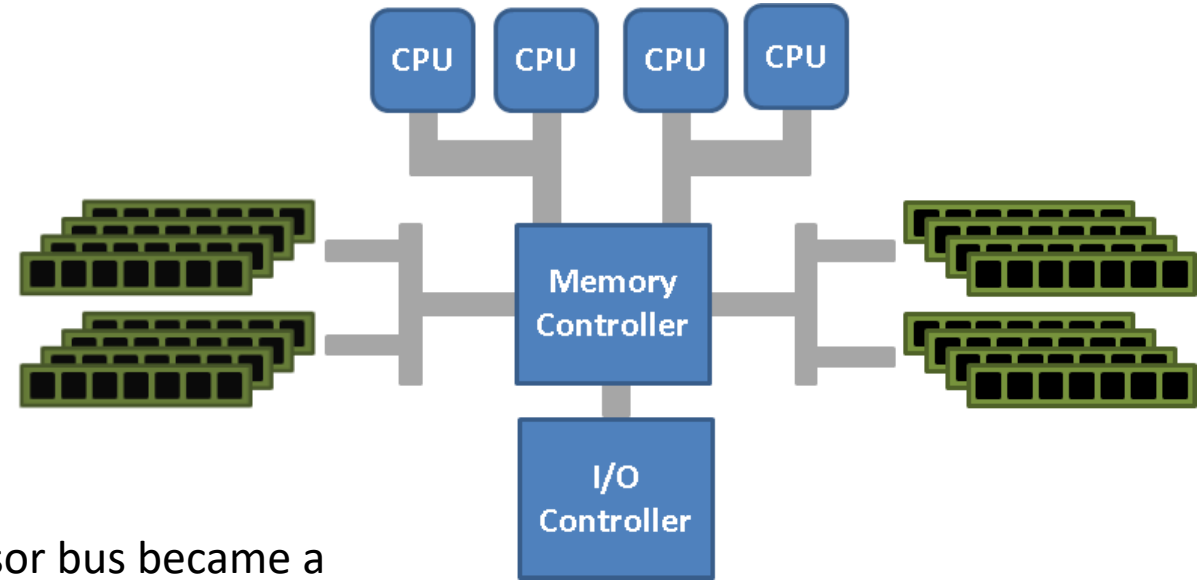
Symmetric Multi-threading (SMT) Hyperthreading



Since 2002

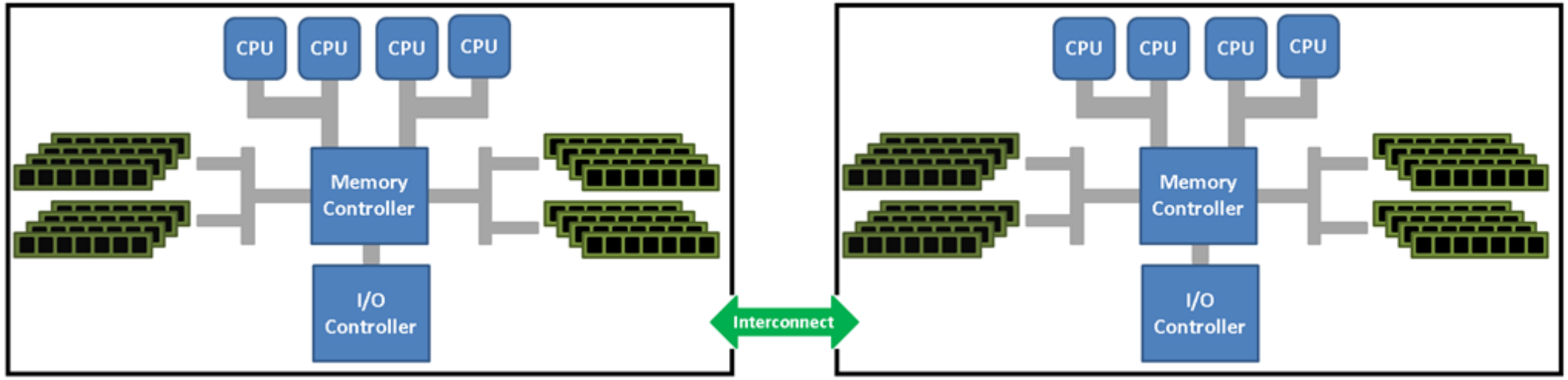
[Hardware insights, Francesco Quaglia]

UMA vs NUMA



In **Symmetric Multiprocessing** (**SMP**) Systems, a single memory controller is shared among all CPUs (**Uniform Memory Access—UMA**). All of the processors have equal access to the memory and I/O in the system.

As more processors were added to the system the processor bus became a limitation to the overall system performance.

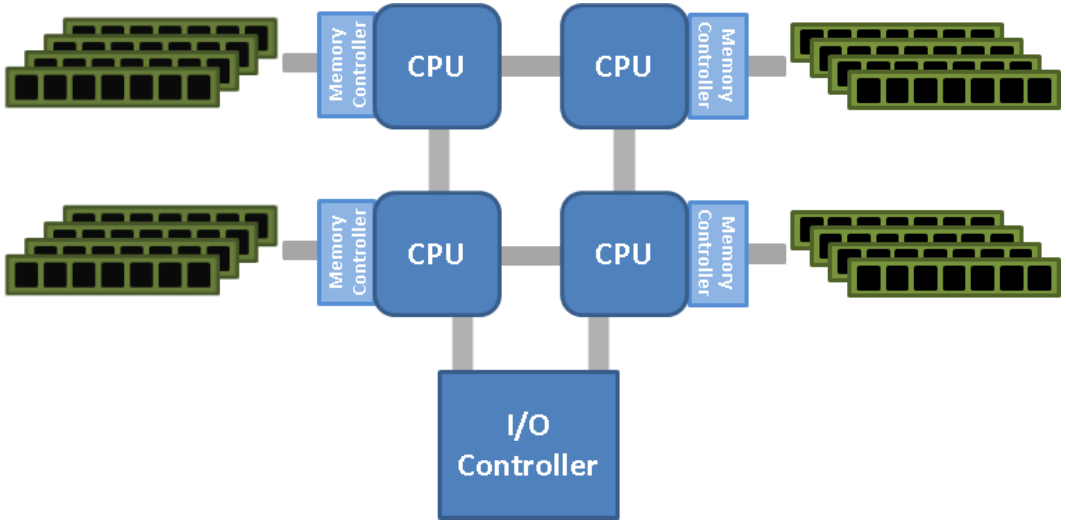


To scale more, **Non-Uniform Memory Architectures** (**NUMA**) implement multiple buses and memory controllers.

The interconnect between the two systems introduced latency for the memory access across nodes.

[Understanding Non-Uniform Memory Access/Architectures (NUMA)]

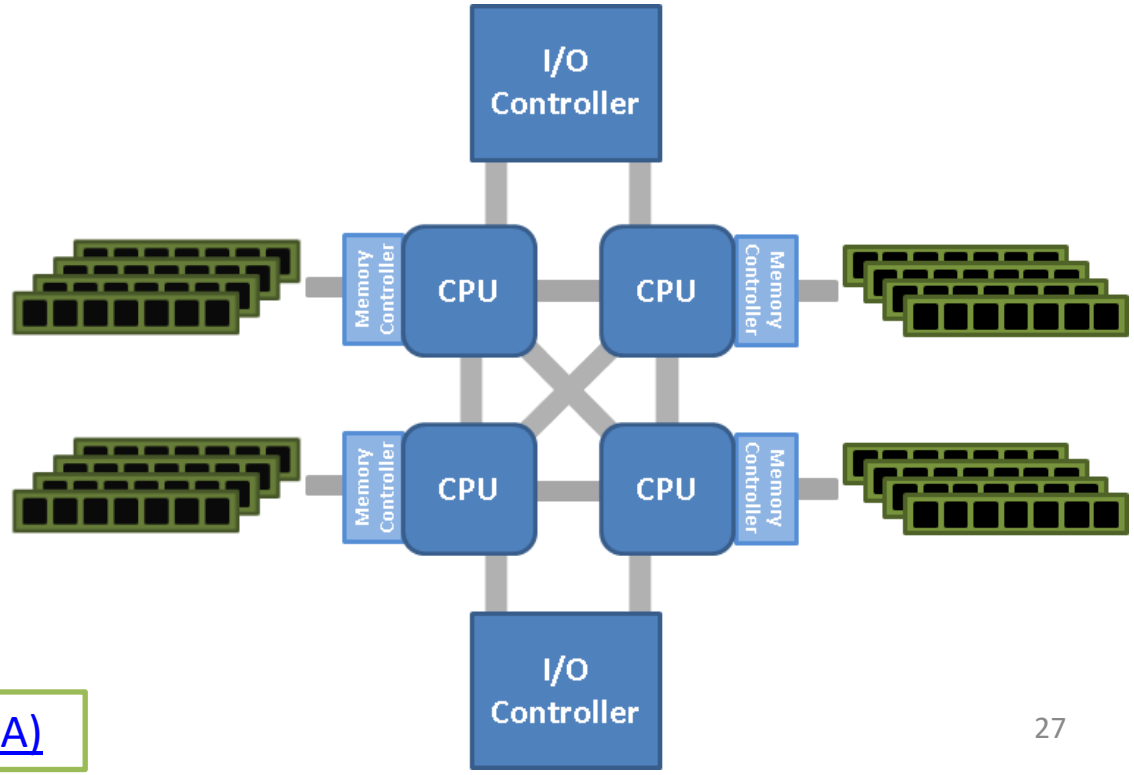
AMD Hyper-Transport (HT)



Where the older SMP architecture had a **separate memory controller**, **newer systems have an** integrated memory controller built into the processor itself, and each processor has its own memory bank. The first processors to introduce an integrated memory controller were the **AMD Opteron** series of processors in early 2003. AMD processors share memory access through **Hyper-Transport (HT) links** between the processors.

Intel Quick-Path Interconnect (QPI)

The Intel QPI interconnects the processors with each other in a similar manner to the AMD HT interconnect. However, the QPI snoop based cache forwarding implementation can return remote data in as little as 2 hops.

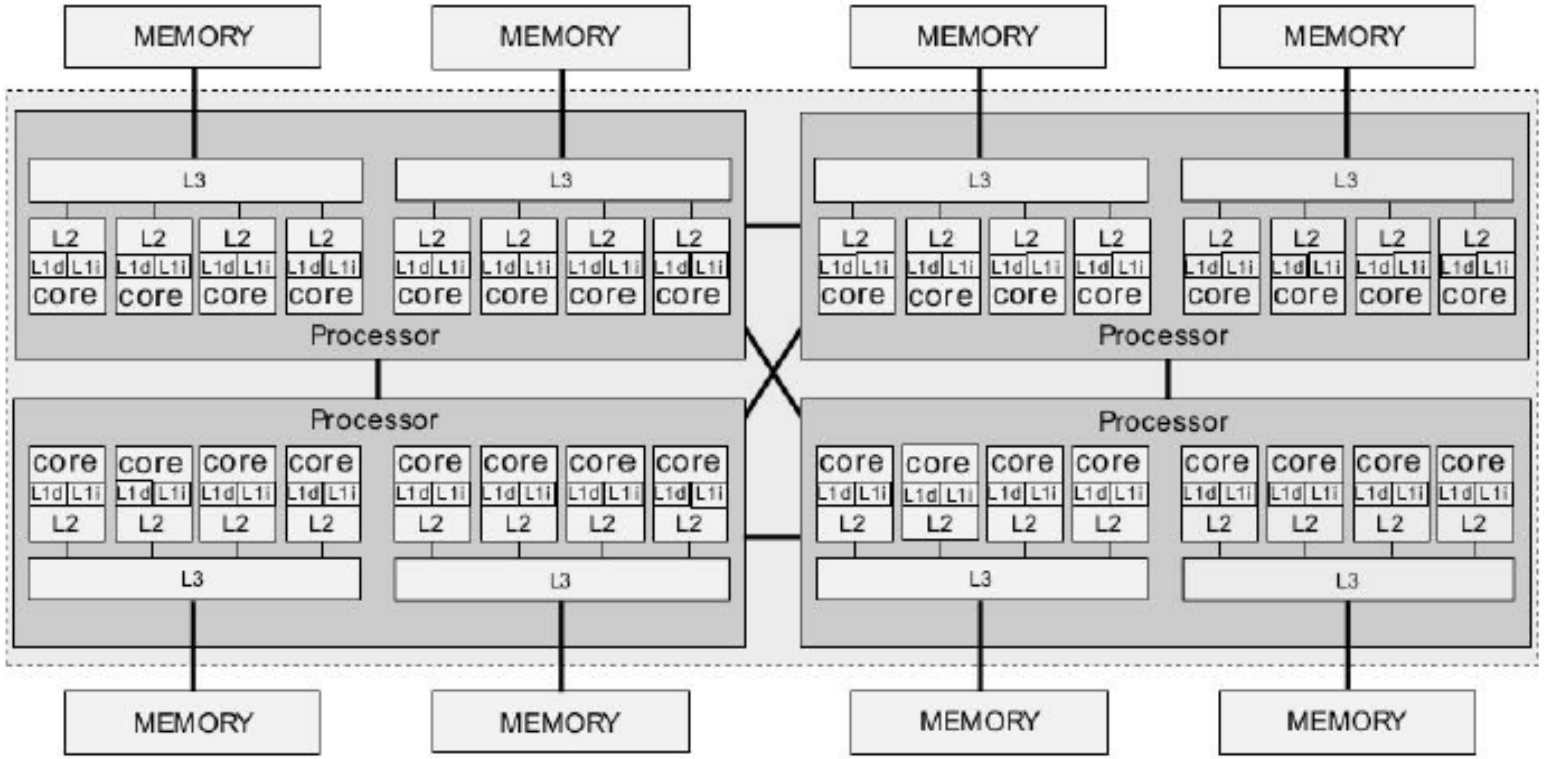


[Understanding Non-Uniform Memory Access/Architectures (NUMA)]

Non-Uniform Memory Access

Each CPU has its own local memory which is accessed faster

- Shared memory is the union of local memories.
- The latency to access remote memory depends on the ‘distance’.



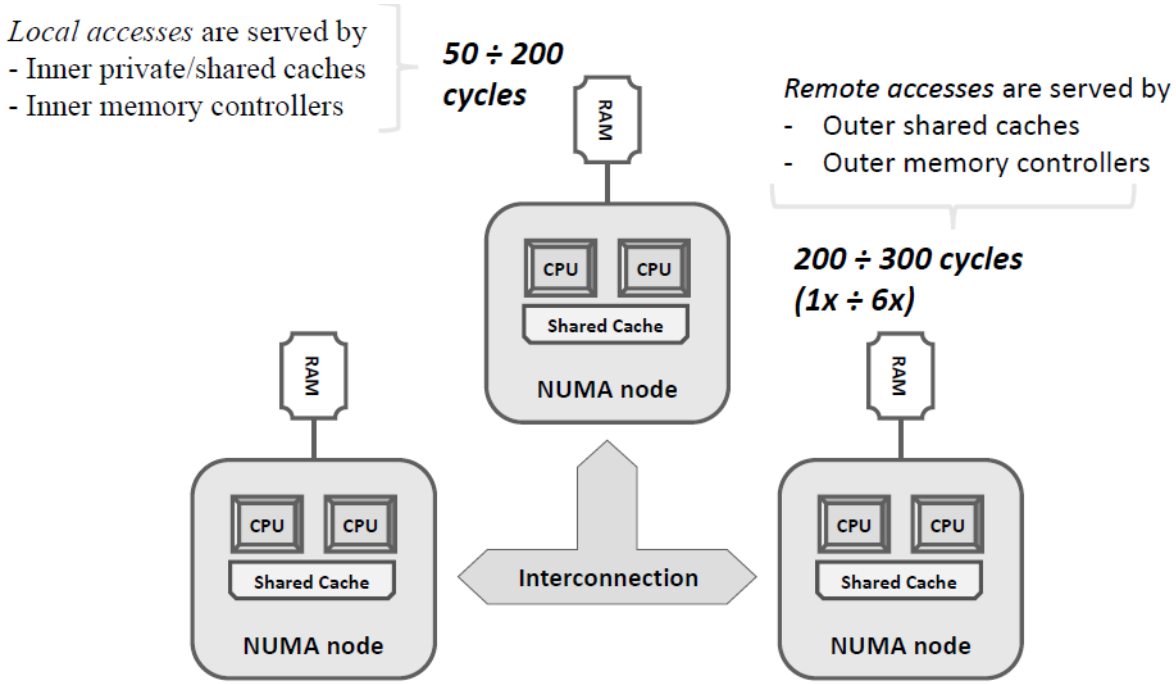
NUMA organization with **four** AMD Opteron 6128 (2010)

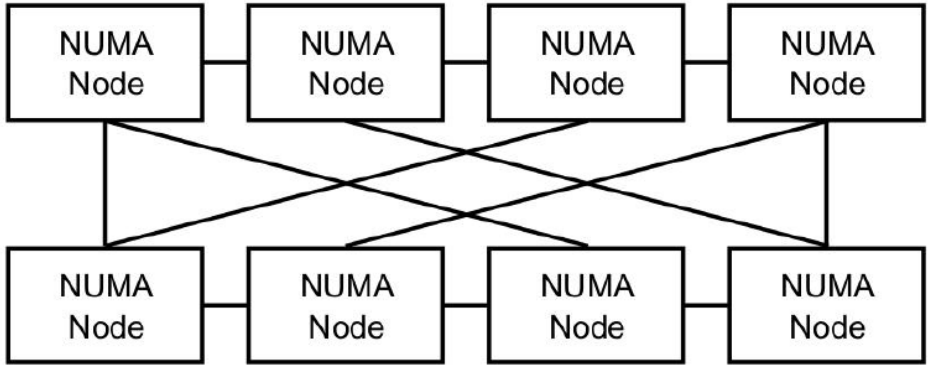
1 AMD Opteron 6128
8 cores (8 threads)
Level 1 cache:
8 x 64 KB instruction cache
8 x 64 KB data cache
Level 2 cache: 8 x 512 KB
Level 3 cache: 2 x 6 MB
Physical memory: 128 GB

A Primer on Modern Hardware Architectures, Alessandro Pellegrini

NUMA latency assymetries

- A processor (made of multiple cores) and the memory local to it form a **NUMA node**.
- There are commodity systems which are not fully meshed: remote nodes can be only accessed with multiple hops.
- The effect of a hop on commodity systems has been shown to produce a performance degradation of even 100%—but it can be even higher with increased load on the interconnect.





Linux is NUMA-aware – Support started in **2004** (Linux 2.6)

Two optimization areas
– **Thread scheduling**
– **Memory allocation**

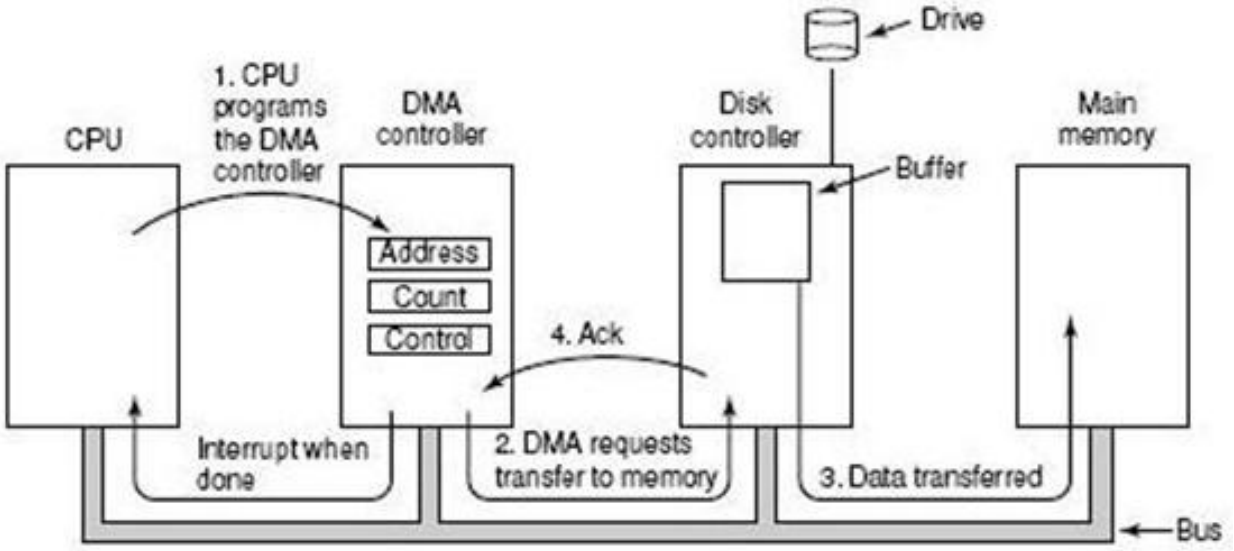
Hardware insights, Francesco Quaglia

A Primer on Modern Hardware Architectures, Alessandro Pellegrini

Direct Memory Access (DMA)

DMA is a **hardware** mechanism that allows the device to **directly transfer data from/to memory** without involving the processor. This significantly increases the transmission efficiency.

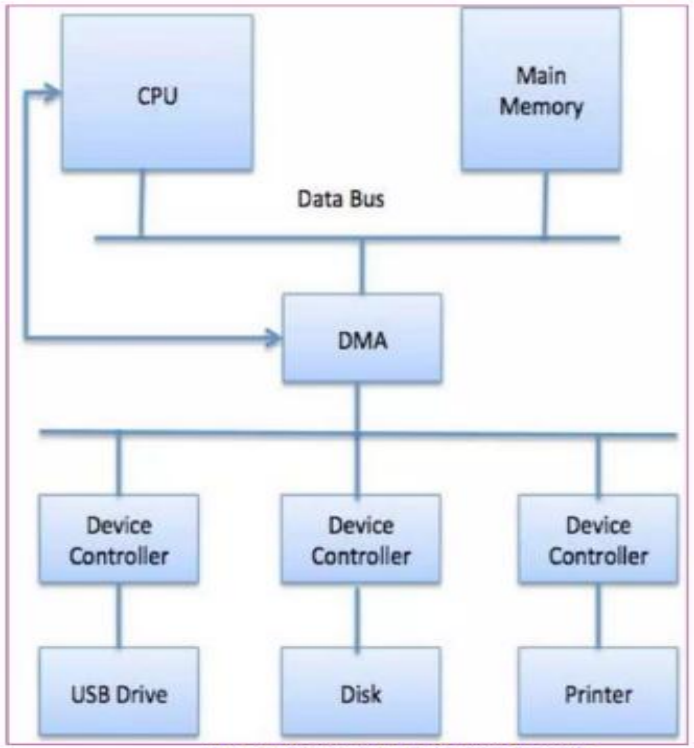
1. The processor supplies to the **DMA controller** the device number, the type of the operation, the memory address and the number of bytes to transfer.
2. DMA **starts** the operation and **arbitrates** the bus while transferring the data.
3. When the transfer is complete the DMA controller **interrupts** the processor.



Direct Memory Access (DMA) (source: Maarten van Steen)

If the DMA buffer occupies **more than one page**, it must be allocated in a consistent area of **physical memory**, because data transmissions on the **ISA** or **PCI system bus** are described by physical addresses. Memory has to be reserved from the **DMA zone**.

DMA must consider the **cache consistency** problem.



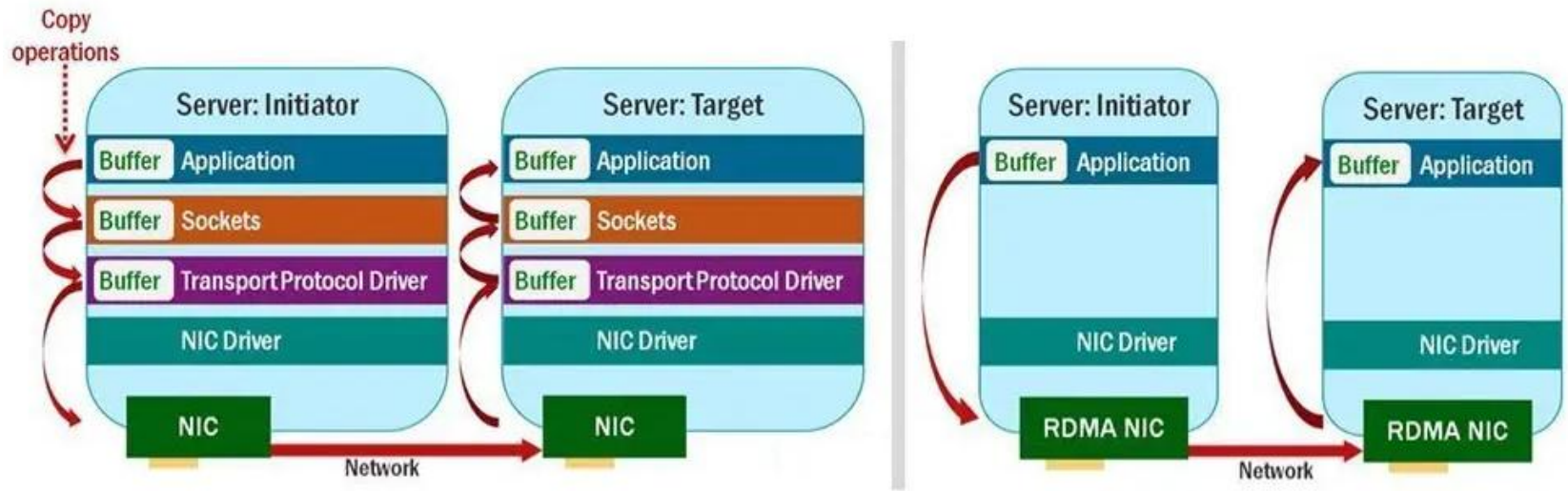
Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) provides direct memory access from the memory of one host to the memory of another host without involving the remote Operating System and CPU, boosting network and host performance with lower latency, lower CPU load and higher bandwidth. In contrast, TCP/IP communications typically require copy operations, which add latency and consume significant CPU and memory resources.

RDMA supports **zero-copy networking** by enabling the **network adapter** to transfer data directly to or from application memory, eliminating the need to copy data between application memory and the data buffers in the operating system. Such transfers require no work to be done by CPUs, caches, or context switches, and transfers continue in parallel with other system operations. When an application performs an RDMA Read or Write request, the application data is delivered directly to the network, reducing latency and enabling fast message transfer.

Where used: High Performance Computing, Machine Learning, Big Data

https://www.teimouri.net/review-whats-remote-direct-memory-access-rdma/



Memory Management Unit (MMU)

- Layout of physical memory
- Memory addressing
- Memory translation and MMU

Additional reading

- Intel 64 and IA-32 Architectures Software Developer Manuals
- What Every Programmer Should Know About Memory (Ulrich Drepper, Red Hat)
- Thread Local Storage (TLS) descriptors in GDT

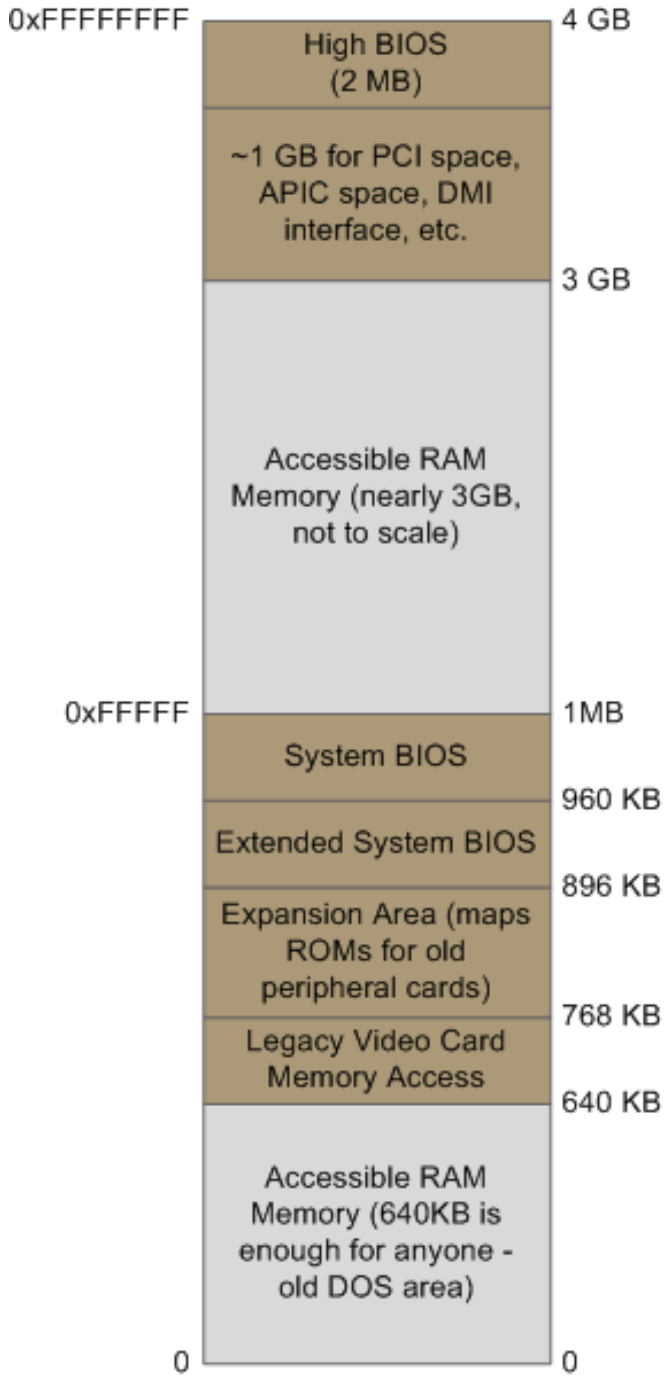
Layout of physical memory – example

Most processor requests are for **RAM**, part for **memory-mapped I/O**, mainly devices such as video cards, PCI cards, as well as flash memory that stores the BIOS.

This **device address mapping** causes a **hole** in the computer memory between 640 KB and 1 MB.

This is why 32-bit operating systems have problems using 4 GB of memory.

The figure shows a **typical layout of the first 4 GB physical memory addresses** in the Intel PC.



Layout of physical memory, 2008
(source: Duarte, Software Illustrated)

On Linux, physical memory mapping can be read from the file **/proc/iomem**. This is how it looks on my old workstation and on duch.mimuw.edu.pl.
(**640 KB** is **0x000a0000**, **1 MB** is **0x00100000**)

```
00000000-0009ffff : System RAM
000a0000-000bffff : Video RAM area
000c0000-000cf7ff : Video ROM
000d1000-000d3fff : Adapter ROM
000f0000-000fffff : System ROM
00100000-3ff73fff : System RAM
00100000-002baebc : Kernel code
002baebd-0037661f : Kernel data
003bc000-0041f57f : Kernel bss
3ff74000-3ff75fff : ACPI Non-volatile Storage
3ff76000-3ff96fff : ACPI Tables
3ff97000-3fffffff : reserved
e8000000-efffffff : 0000:00:00.0
f0000000-f7ffffff : PCI Bus 0000:01
f0000000-f7ffffff : 0000:01:00.0
fcf00000-fcffffff : PCI Bus 0000:02
fcfe0000-fcffffff : 0000:02:0c.0
fcfe0000-fcffffff : e1000
fd000000-feafffff : PCI Bus 0000:01
fd000000-fdffffff : 0000:01:00.0
fea00000-fea1ffff : 0000:01:00.0
febff900-febff9ff : 0000:00:1f.5
febff900-febff9ff : Intel ICH5
febffa00-febffbff : 0000:00:1f.5
febffa00-febffbff : Intel ICH5
febffc00-febfffff : 0000:00:1f.1
fec00000-fec0ffff : reserved
fecf0000-fecf0fff : reserved
fed20000-fed8ffff : reserved
fee00000-fee0ffff : reserved
ffa80800-ffa80bff : 0000:00:1d.7
ffa80800-ffa80bff : ehci_hcd
ffb00000-ffffffff : reserved
```

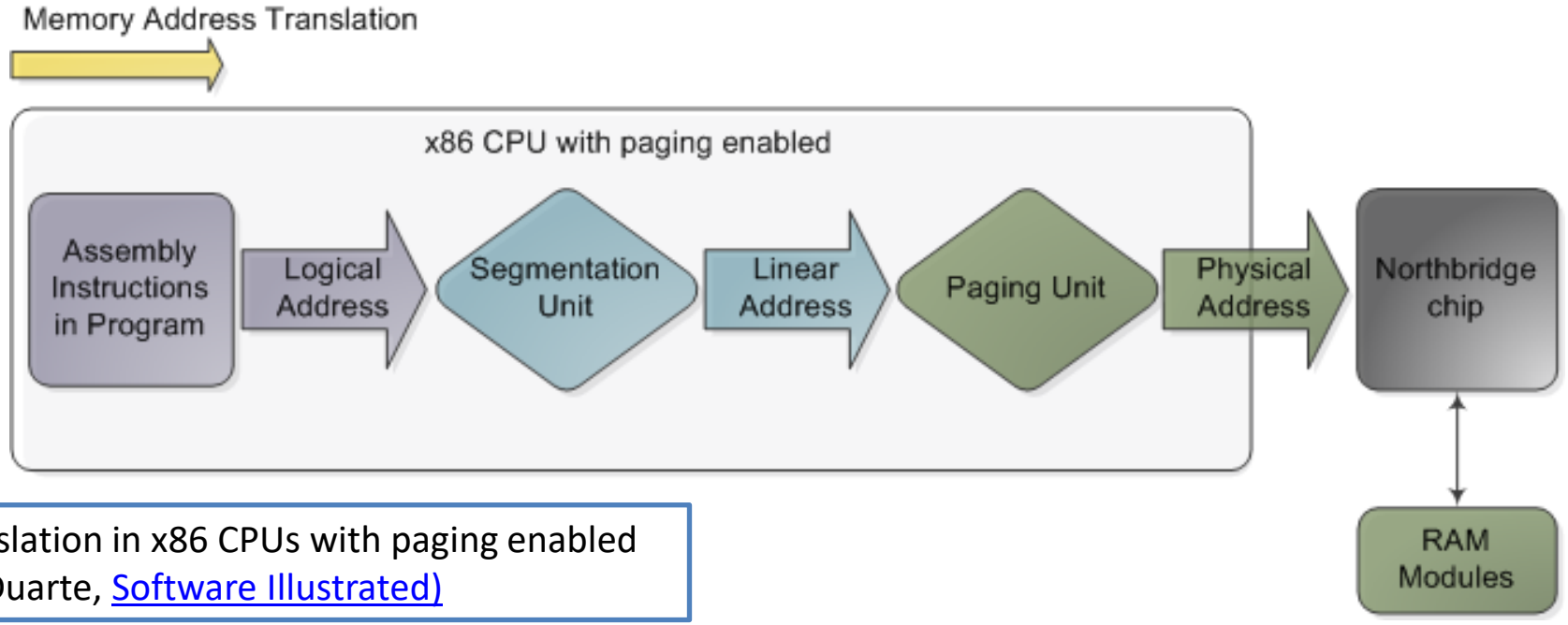
Do you see the difference?

```
[jmd@duch ~]$  cat /proc/iomem
00000000-00000000 : reserved
00000000-00000000 : System RAM
00000000-00000000 : reserved
00000000-00000000 : PCI Bus 0000:00
00000000-00000000 : Video ROM
00000000-00000000 : reserved
  00000000-00000000 : System ROM
00000000-00000000 : System RAM
00000000-00000000 : reserved
00000000-00000000 : PCI Bus 0000:00
  00000000-00000000 : 0000:00:02.0
  00000000-00000000 : 0000:00:0b.0
  00000000-00000000 : 0000:00:02.0
  00000000-00000000 : 0000:00:03.7
    00000000-00000000 : ehci_hcd
  00000000-00000000 : 0000:00:04.0
  00000000-00000000 : 0000:00:05.0
  00000000-00000000 : 0000:00:08.0
  00000000-00000000 : 0000:00:09.0
  00000000-00000000 : 0000:00:0a.0
  00000000-00000000 : 0000:00:0b.0
  00000000-00000000 : 0000:00:02.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:04.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:05.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:06.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:07.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:08.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:09.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:0a.0
    00000000-00000000 : virtio-pci-modern
  00000000-00000000 : 0000:00:0b.0
    00000000-00000000 : virtio-pci-modern
00000000-00000000 : IOAPIC 0
00000000-00000000 : Local APIC
00000000-00000000 : reserved
00000000-00000000 : reserved
00000000-00000000 : System RAM
  00000000-00000000 : Kernel code
  00000000-00000000 : Kernel data
  00000000-00000000 : Kernel bss
00000000-00000000 : PCI Bus 0000:00
```

Memory addressing

Memory addressing depends on the **hardware**. **80x86** microprocessors distinguish three types of addresses:

- **Logical address** – operated at the level of **machine language instructions**. It is related to **segmentation** available in this architecture. Each logical address consists of a segment number and a segment offset.
- **Linear (virtual) address** – is a 32-bit number that allows to address up to 4 GB.
- **Physical address** – address recognized by the memory module. Physical addresses are in the form of a 32-bit or 36-bit number.



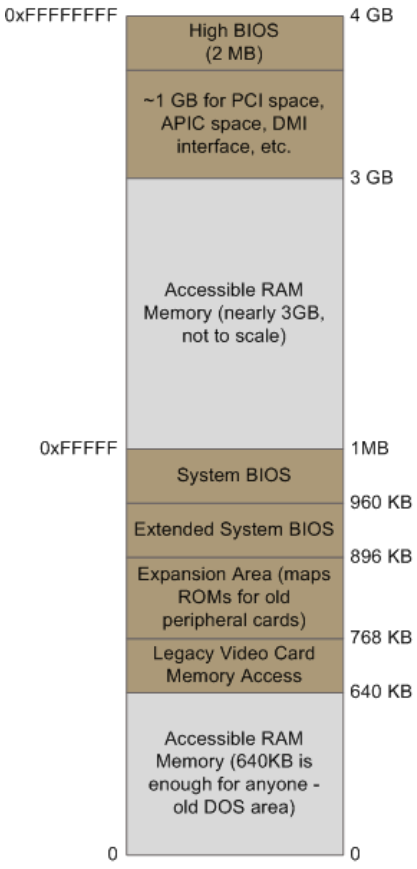
Memory address translation in x86 CPUs with paging enabled (source: Duarte, Software Illustrated)

Address translation and MMU

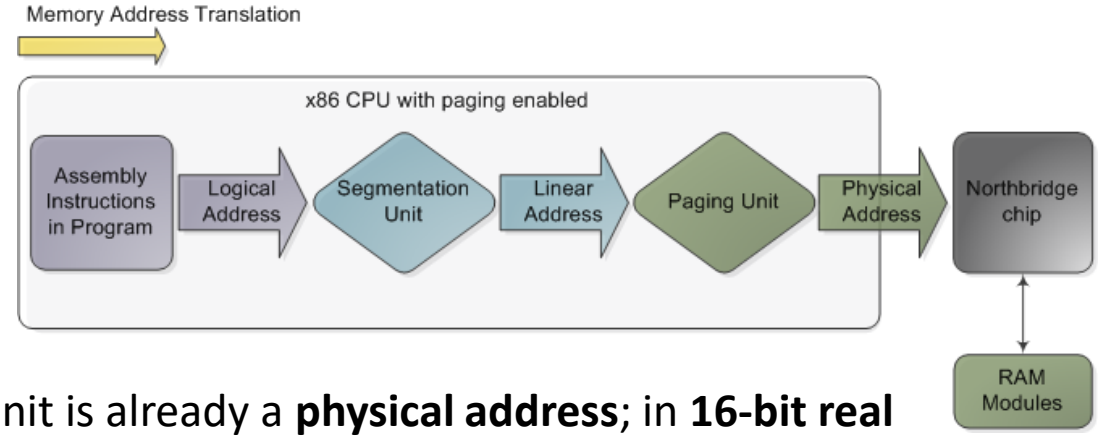
The translation of addresses is done by hardware **Memory Management Unit** (**MMU**).

The **translation** of **logical addresses** into **physical addresses** depends on the **processor mode**.

- In **real mode**, the CPU can address only **1 MB** of physical RAM.
- When the CPU works in **32-bit protected mode**, it can physically address only **4 GB** of memory (unless it uses the so-called physical address extension). Because the upper 1 GB is mapped to devices connected to the motherboard, only **3 GB** of RAM remain effectively.
- When the CPU works in **64-bit protected mode**, it can physically address 64 GB (hardly any motherboards deliver enough RAM). In this mode, you can use physical addresses above the RAM available in the system to reach RAM areas corresponding to the physical addresses stolen by devices connected to the motherboard.



Segmentation



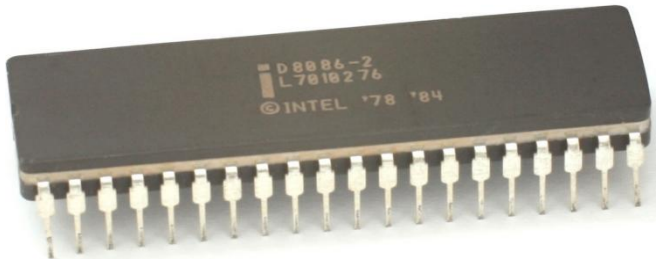
When **paging** is **turned off**, the output from the segmentation unit is already a **physical address**; in **16-bit real mode** that is always the case.

The original 8086 (1978) had 16-bit registers. This allowed code to work with 2^{16} bytes or 64 KB of memory. To increase the size of the available address space, without increasing the size of registers and instructions, **segment registers** were introduced to allow **switching** between different blocks (**segments**) of 64 KB size.

There were **four segment registers**: for **stack** (**ss**), for **program code** (**cs**), for **data** (**ds**, **es**). There are also two **general-purpose** segment registers: **fs** and **gs**.

Nowadays **segmentation** is still **present** and is **always enabled** in x86 processors.

Each instruction that touches memory implicitly uses a **segment register**. Segment registers store **16-bit segment selectors**.



https://en.wikipedia.org/wiki/X86

https://en.wikipedia.org/wiki/X86_memory_segmentation

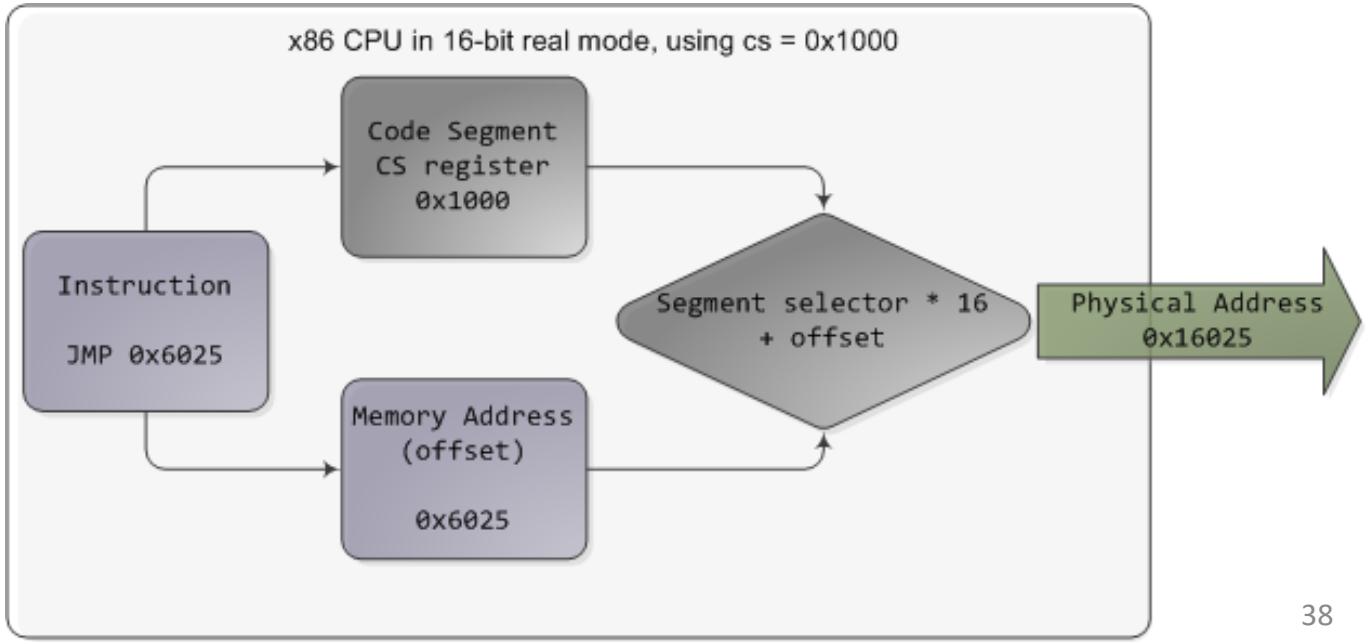
16-bit real mode

Though segmentation is always on, it works **differently** in **real mode** versus **protected mode**.

In **real mode**, such as during **early boot**, the **segment selector** is a **16-bit number** specifying the **physical memory address** for the **start of a segment**.

This number must be scaled. Intel made the decision to multiply the segment selector by only $\mathbf{2^4}$ (or 16), which limits memory to about **1 MB** and **complicates translation** (many **segment:offset** combinations resolve to the same physical address). The resulting physical address has **20-bits**.

The example shows a jump instruction where **cs** registry contains 0x1000.



Real mode segmentation (source: Duarte, Software Illustrated)

32-bit protected mode

https://en.wikipedia.org/wiki/I386

In **32-bit protected mode** (1982), a segment selector is an **index** into a **table** of 8-byte **segment descriptors**. Segment descriptors are stored in two tables: **Global Descriptor Table** (**GDT**) and **Local Descriptor Table** (**LDT**).

The **TI bit** in a segment selector is 0 for the **GDT** and 1 for the **LDT**, while the **index** specifies the desired segment within the table. Each **CPU** (or **core**) contains a register called **gdtr**, which stores the **linear memory address of the first byte in the GDT**.

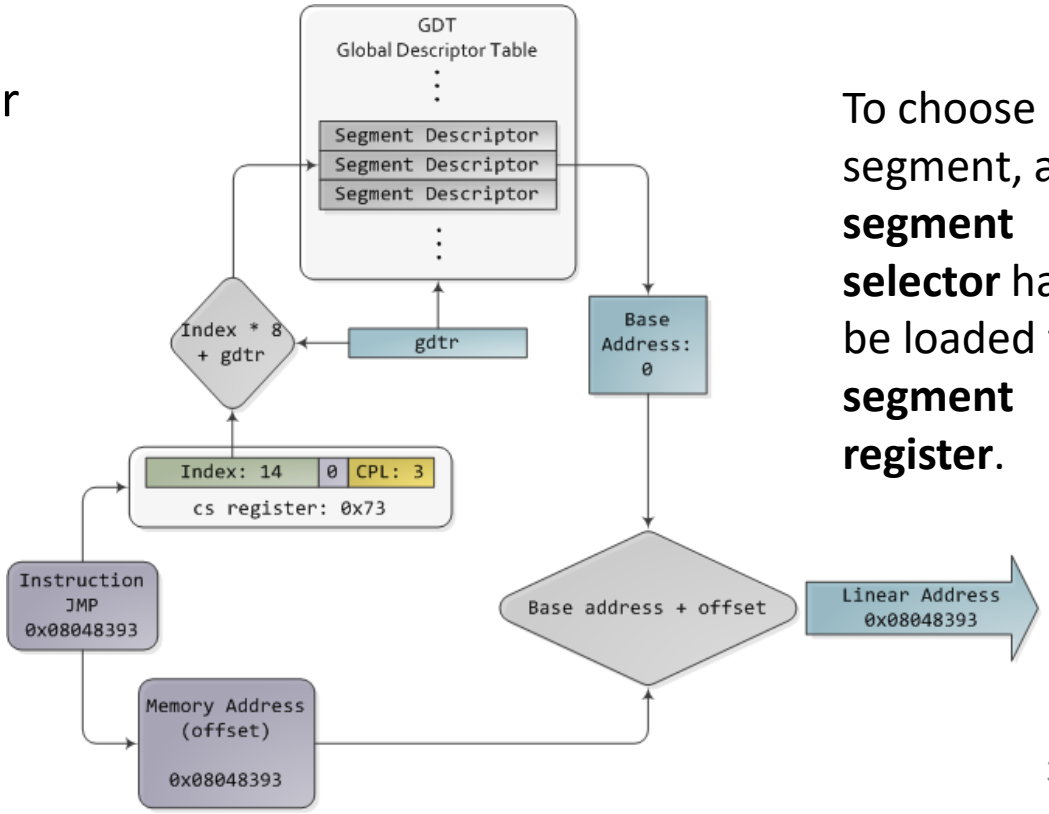
The **base address** in segment descriptor is a 32-bit linear address pointing to the beginning of the segment.

The **limit** specifies how big the segment is.

Adding the base address to a logical memory address yields a **linear address**.

The jump instruction in 32-bit protected mode.

cs contains **code segment selector**.



To choose a segment, a **segment selector** has to be loaded to a **segment register**.

Segmentation in protected mode (source: Duarte, Software Illustrated)

GDT and LDT

Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

Global Descriptor Table (GDT) (source: Bovet, Cesati, Understanding the Linux Kernel)

There is **one copy** of **GDT** for **each processor**. All copies have identical content except for a few items. Each processor has its own **TSS** segment (**Task State Segment**). Several items depend on the **process** that is performed on a given processor (**LDT** and **TLS** segment descriptors).

TSS is used to save the contents of the processor registers. These segments are arranged one by one in the **init_tss array**. Each TSS has 236 bytes and cannot be accessed in user mode.

TLS (**Thread Local Storage**) allows **threads** within one program to use up to three segments containing local data for each thread. The system functions **set_thread_area()** and **get_thread_area()** are used to create and release a TLS segment for a **running process**.

Most user mode applications do not make use of LDT, thus the kernel defines default LDT to be shared by most processes. In some cases, however, processes may require to set up their own LDT (e.g. applications, like Wine, that execute segment-oriented MS Windows applications).

Basic flat model

After boot, each CPU has its **own copy** of the **GDT**:

- the **layout** of the GDT is specified in arch/x86/include/asm/segment.h,
- and its **instantiation** in arch/x86/kernel/cpu/common.c.

When the CPU is in **32-bit mode**, registers and instructions can address the **entire linear address space**, it is enough to set the base address to 0 and treat the logical address as a linear address. Intel calls it **basic flat model**.

Basic flat model is equivalent to **disabling segmentation** when it comes to translating memory addresses.

All processes executed in user mode use the same pair of segments **user code segment** and **user data segment**.

Similarly, all kernel mode processes use the same pair of segments **kernel code segment** and **kernel data segment**.

All of these segments have a base address set to 0 and a limit of 2^{32}-1. This means that **all processes**, both in **user mode** and in **kernel mode**, can use **the same logical addresses**.

64-bit flat linear address space

Since coinciding logical and linear addresses are simpler to handle, they became **standard**, such that **64-bit mode enforces a flat linear address space** (2003).

Except in unusual cases, segmentation won't change the resulting physical address in 64 bit mode (segmentation is just used to store traits like the current privilege level, and enforce features like SMEP – Supervisor Mode Execution Prevention which can be used to prevent supervisor mode from unintentionally executing user-space code).

One well known "unusual case" is the implementation of **Thread Local Storage** by most compilers on x86, which **uses the fs and gs segments to define per logical processor offsets into the address space**. Other segments can not have non-zero bases, and therefore cannot shift addresses through segmentation.