NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture 2: Resources and computer architecture

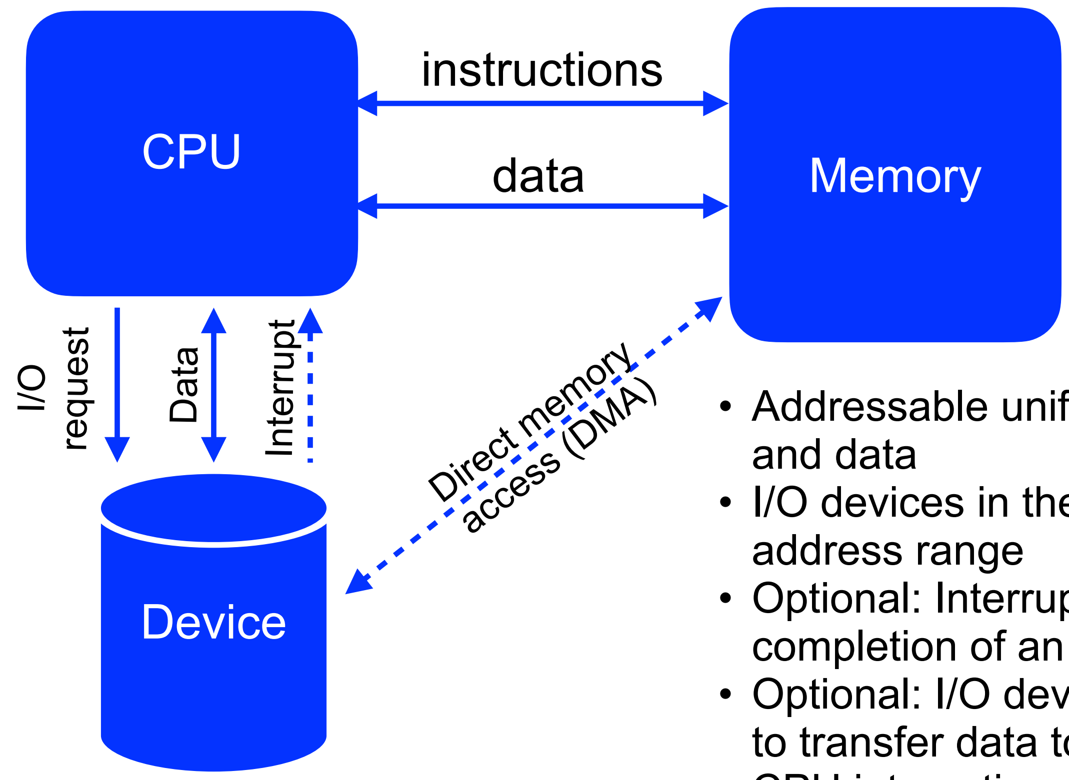
Michael Engel

Overview

- Structure of a typical computer system
  - Basic elements
  - Instruction execution
- From von Neumann to modern computers
  - Memory hierarchy
  - Multiprocessing
  - Communication
  - Heterogeneous systems: GPGPUs
- Non-functional properties
  - Security and virtual memory

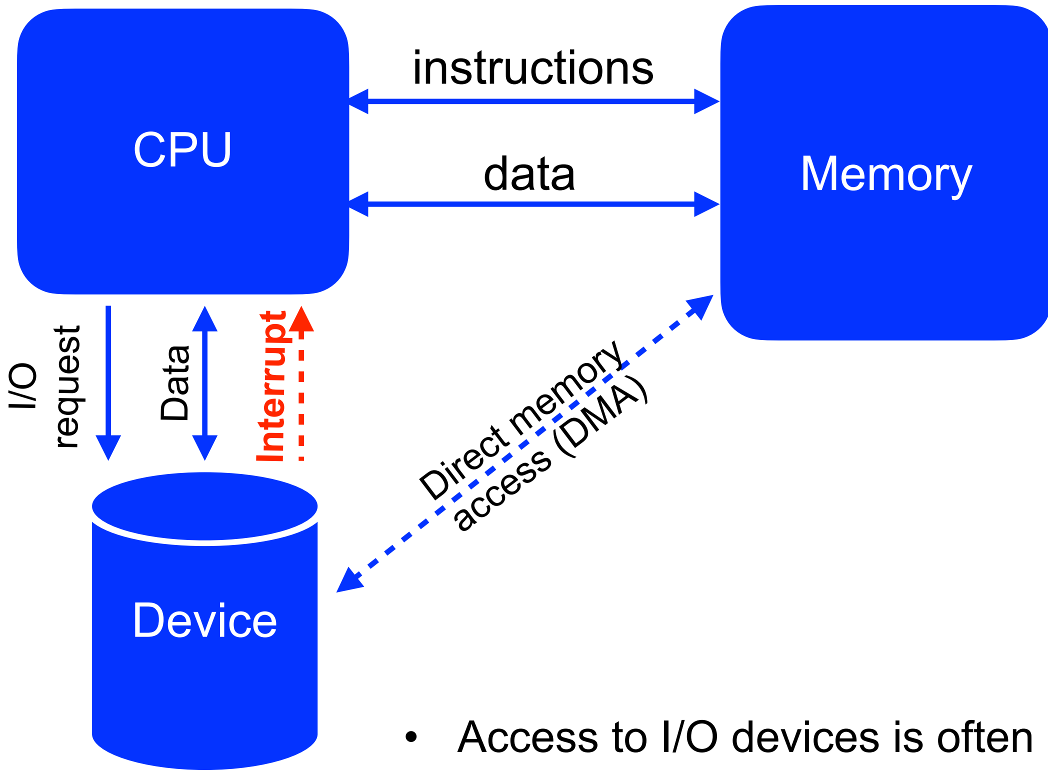
Computers as they are no more

- The typical diagram of a *von Neumann*-style computer system in an introductory course of computer architecture [1]
(this diagram only models very simple microcontrollers today):



- Addressable unified memory for code and data
- I/O devices in the same or a different address range
- Optional: Interrupts notify CPU of the completion of an I/O operation
- Optional: I/O devices can use DMA to transfer data to memory without CPU interaction

Asynchronous execution: interrupts



Polling:

```
write(device, command)
while (not ready(device)) {
   // just wait and waste time!
}
read(device, data)
```

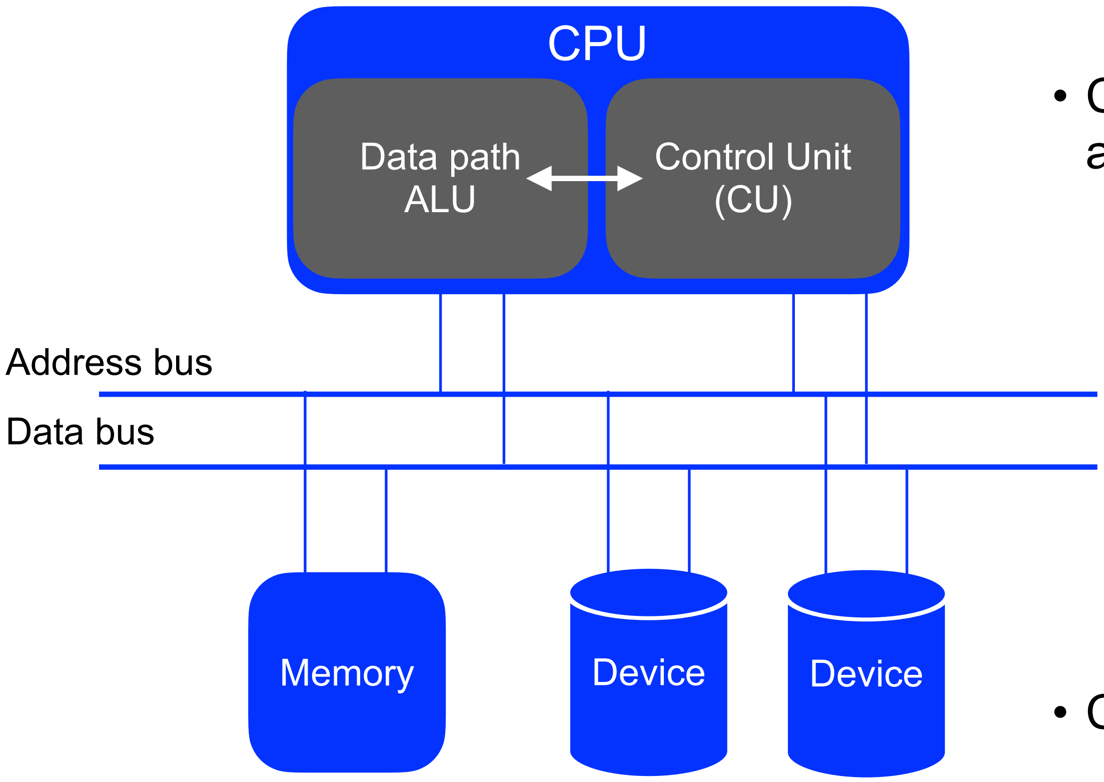
Interrupt driven:

```
write(device, command)
// do something else....................
↯ Interrupt:
   ➛ read(device, data)
```

- Access to I/O devices is often slow
  - **Polling** sends a command and then waits until the device returns data
- With **interrupts**, the device notifies the program when data is ready
  - This changes the **control flow** the CPU executes!
  - More complex to develop software for

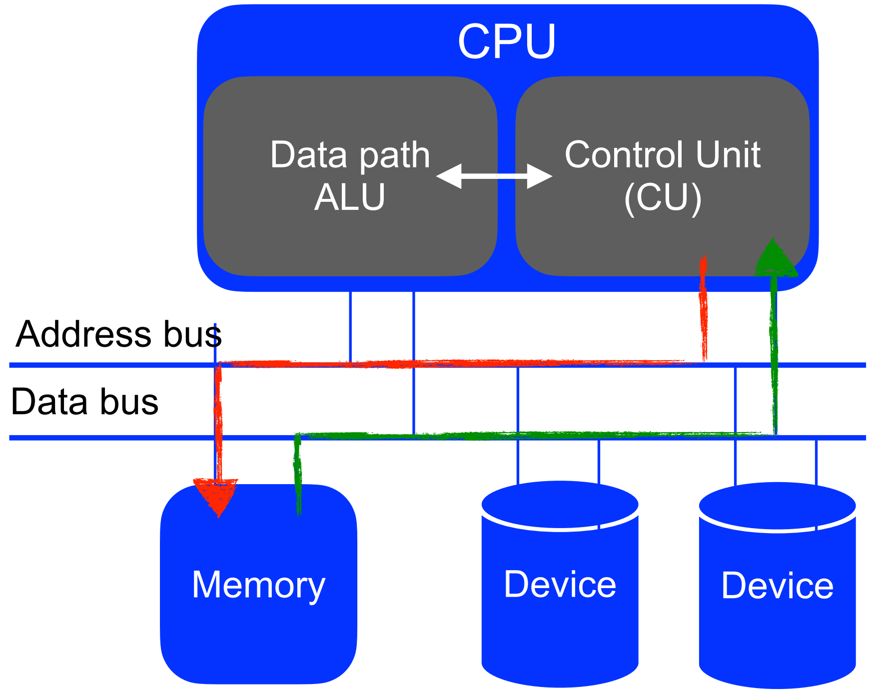
Computers as they are no more

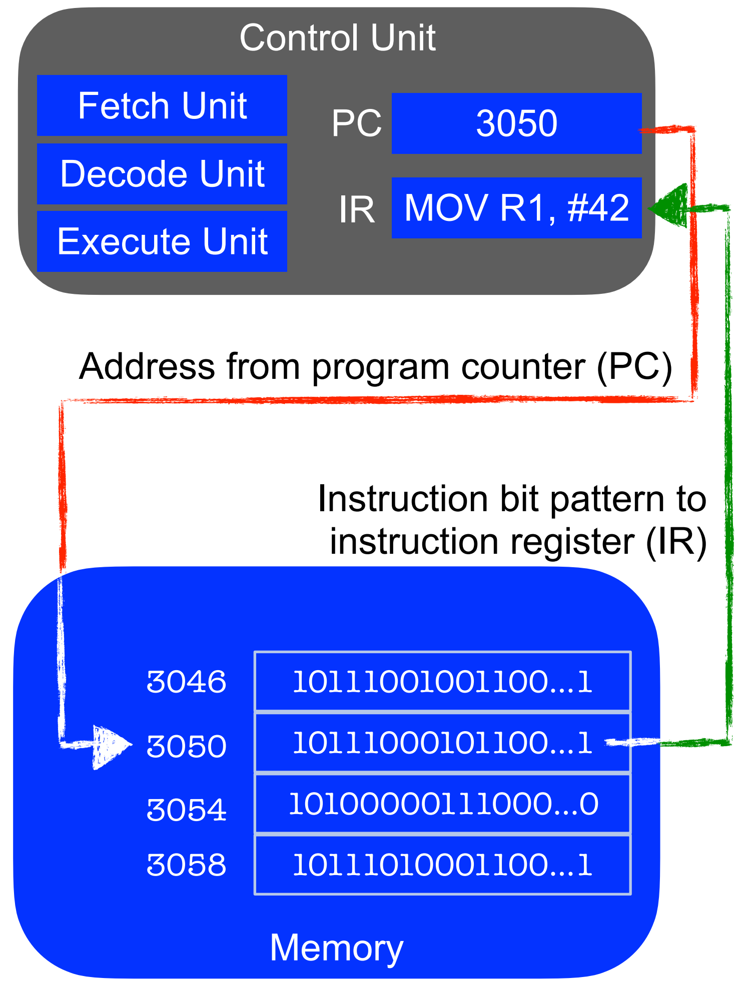
- Going a bit more into details:



- Components of the computer are connected by *buses*
  - Address bus: identify component
  - Data bus: transfer information
  - Control bus: metainformation (read/write, interrupt, …) – not shown here
- CPU has control over the bus
  - Exception: DMA

Instruction execution

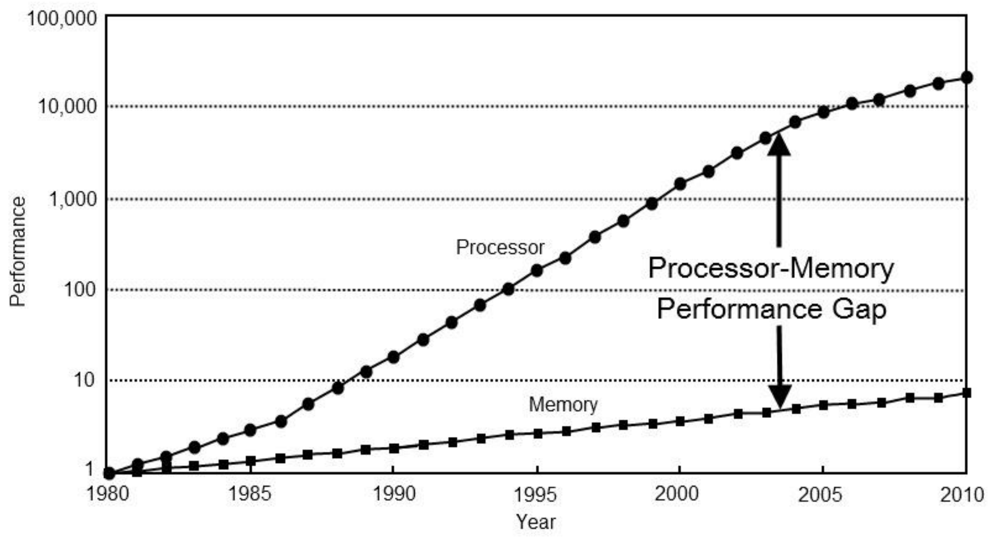





```
PC = <reset address>   // initialize PC
IR = memory[PC]        // fetch first instruction
haltFlag = false
while (not haltFlag) {
        execute(IR)        // execute
        PC = PC + 4        // address of next instr.
        IR = memory[PC]  // fetch it!
}
```

Getting a bit more real

- Simple model of execution only works efficiently if the speed of memory = speed of the CPU
  - This was the case until ca. 1980
- Memory speed only improved ~6%/year
- Today: “memory gap:
  - CPU speed ~ 10,000x faster, but memory speed only ~ 10x



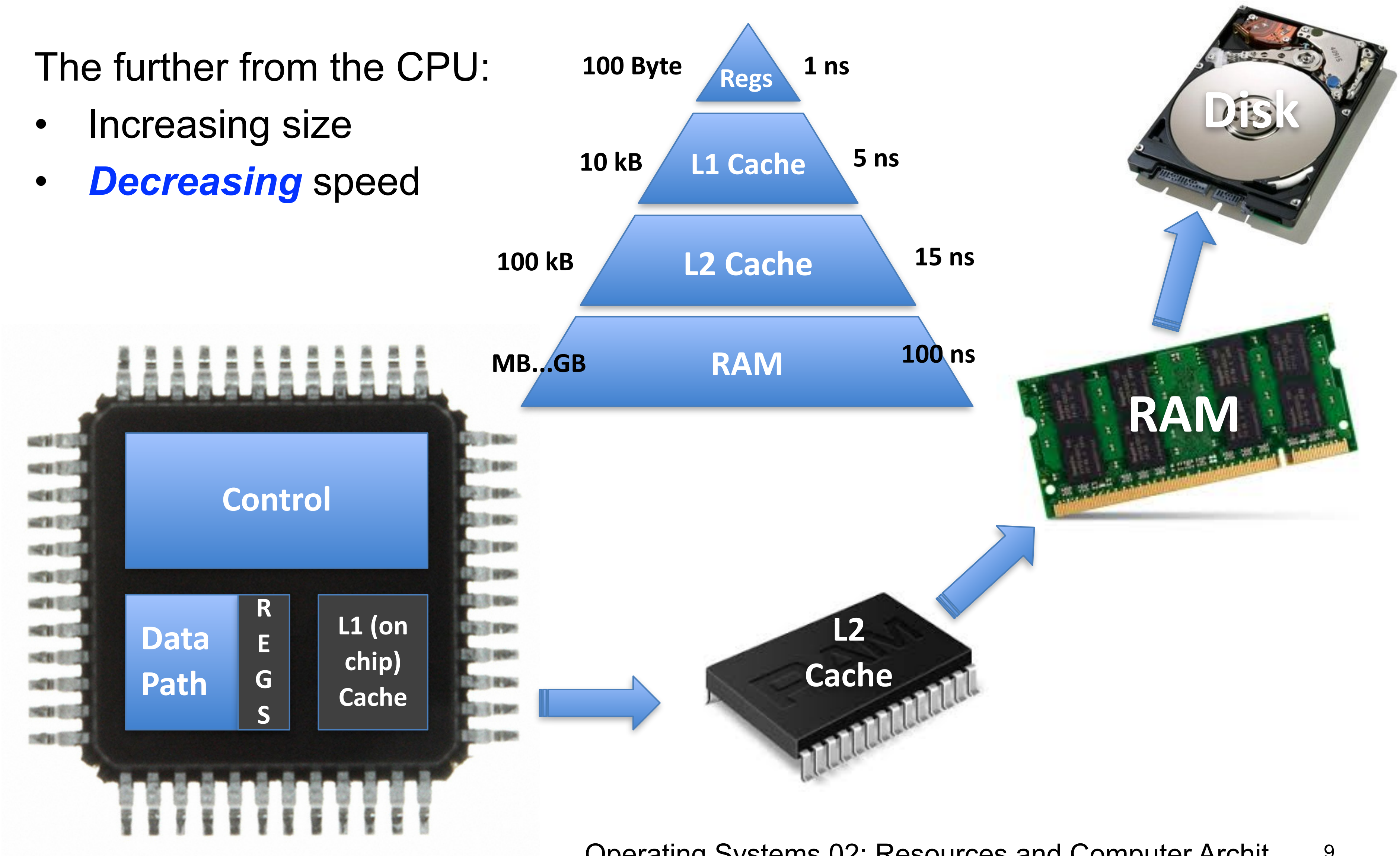
Introducing a memory hierarchy

- Idea: introduce ***caches***
  - small, but fast intermediate levels of memory
- Caches can only hold a ***partial copy*** of the whole memory
  - Unified caches vs. separate instruction and data caches
  - Expensive to manufacture (➛ small)
  - Later: introduction of **multiple levels of cache** (L1, L2, L3…)
    - Each one bigger but slower than the previous one
- Caches work efficiently due to *locality principles* [2]:
  - temporal locality: a program accessing some part of memory is likely to access the same memory soon thereafter
  - spatial locality: a program accessing some part of memory is likely to access nearby memory next

Introducing a memory hierarchy

The further from the CPU:

- Increasing size
- ***Decreasing*** speed



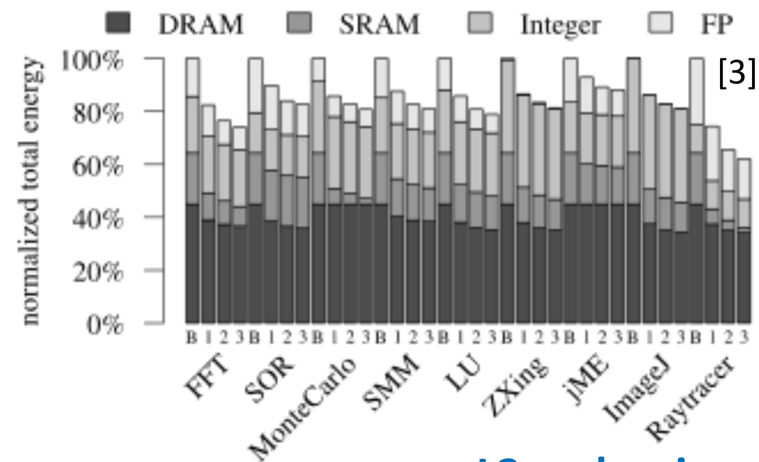
Memory impact: non-functional properties

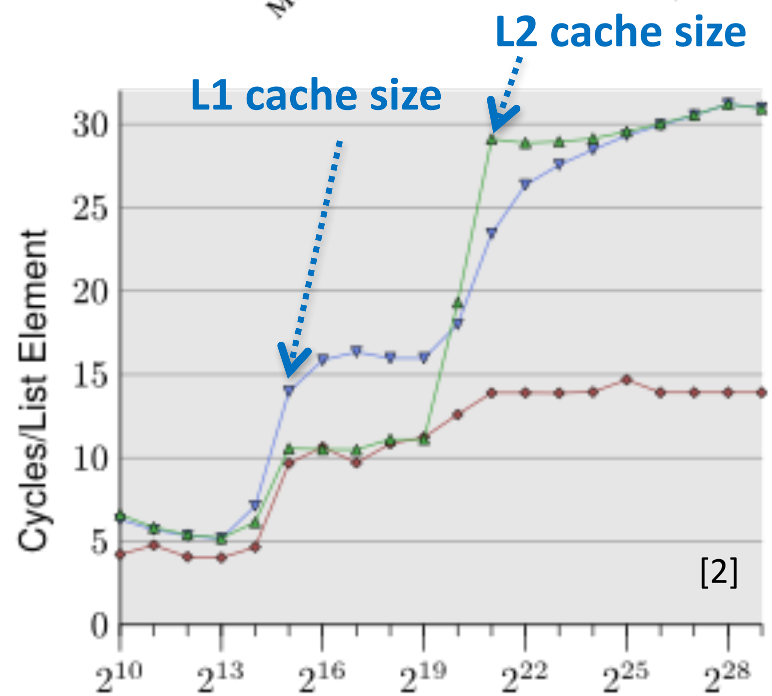
Memory has a large influence on non-functional properties of a system

- Average, best, and worst case performance, throughput and latencies
- Power and energy consumption
- Reliability and security

Non-functional properties depend on many parameters of memory, e.g.

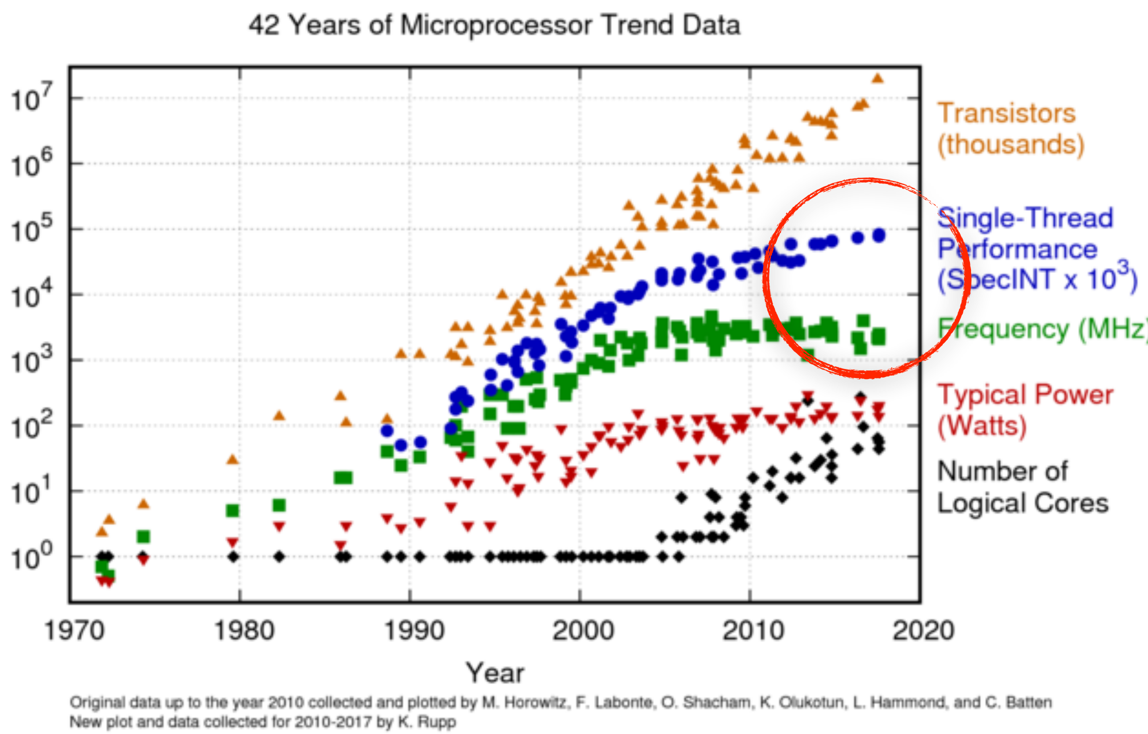
- Cache architecture
- Memory type
- Alignment and aliasing of data





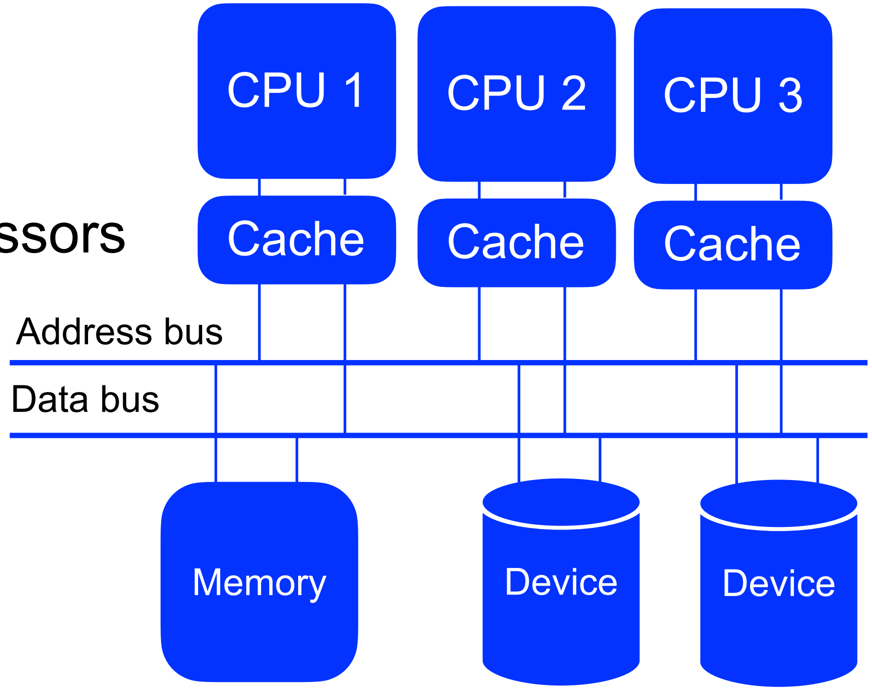
When one processor is not enough

- Moore's Law (1965) [4]:
  - observation that the ***number of transistors*** in a dense integrated circuit (IC) ***doubles about every two years***
  - Accordingly, increase in CPU speed due to smaller semiconductor structures
- This development is hitting physical limitations
  - CPU frequencies "stuck" at ~3 GHz
  - Energy consumption is additional limiting factor



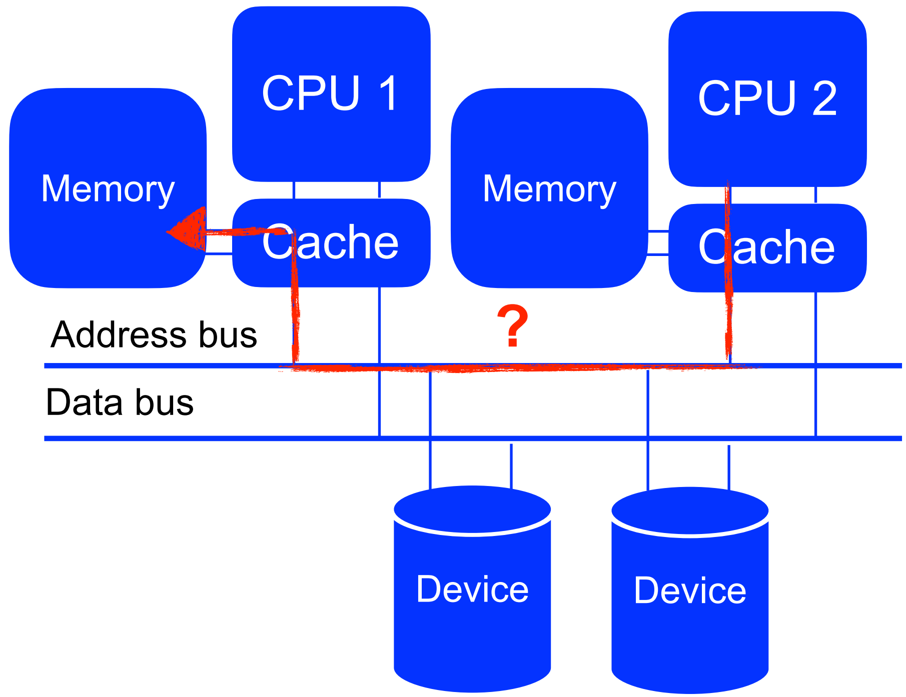
When one processor is not enough

- What can we do with all these transistors?
  - Bigger caches – energy hungry and prone to faults!
  - ➢ ***Put more processors on a chip!***
    - Earlier high-end systems already used multiple separate processor chips
- Old as well as new problems:
  - Memory **throughput** now has to satisfy demands of ***n*** processors
  - **Software** now has to support **execution on multiple processors**!
  - **Caches** need to be **coherent** so they hold the same copies of main memory data

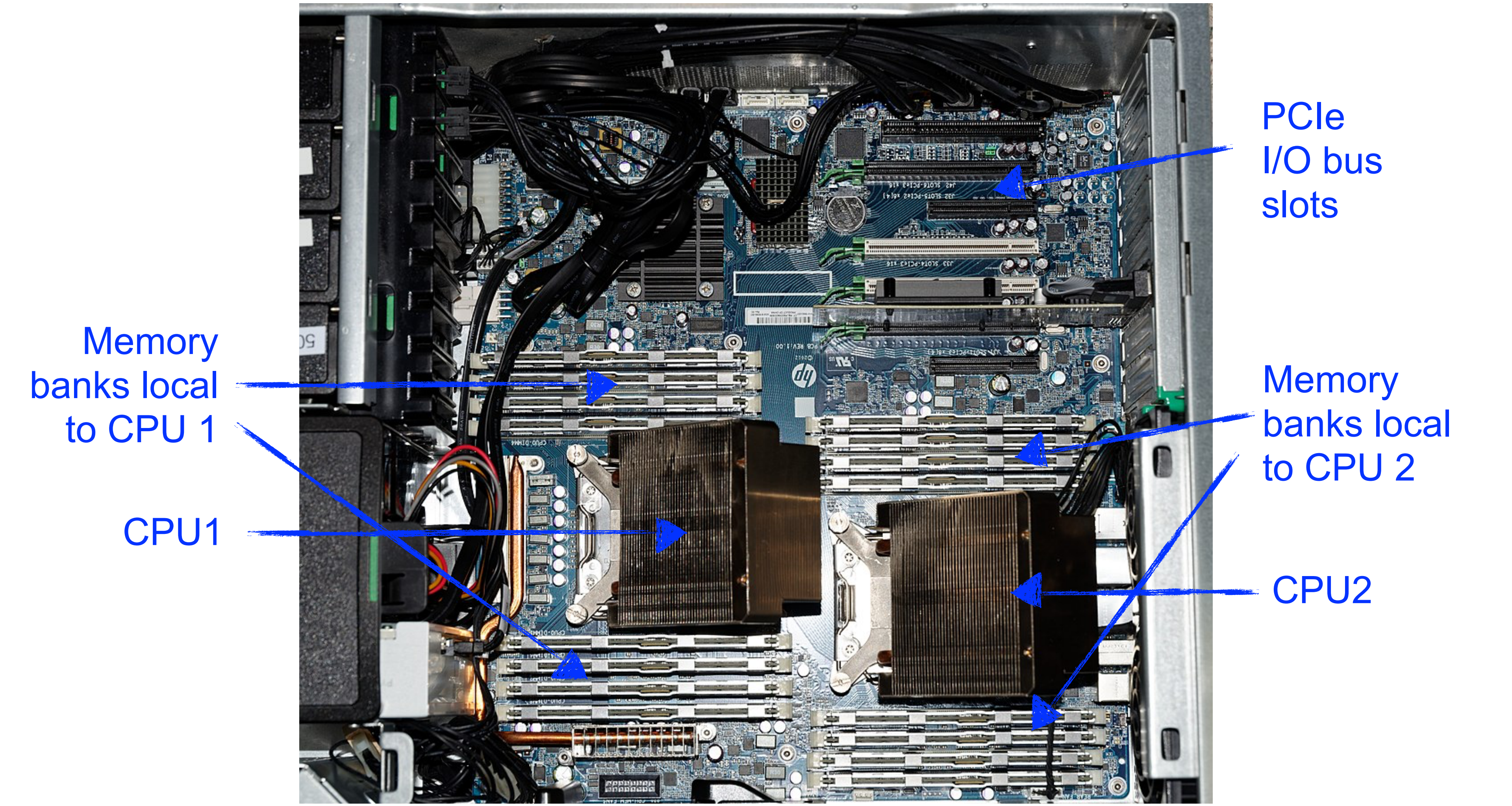


More processors, more memories

- Memory ***throughput*** now has to satisfy demands of ***n*** processors
  - Provide each processor with its own main memory!
  - **NUMA**
    "non unified memory architecture"
- And new problems show up:
  - How to access data in another CPU's memory?
  - Who decides which CPU is allowed to use the bus?
  - Is a common bus still efficient?



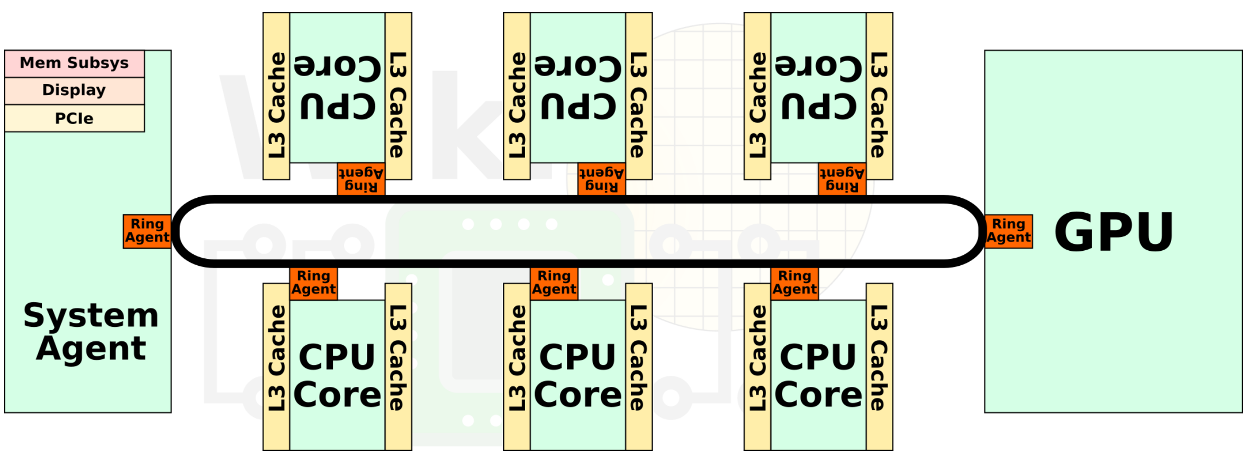
A NUMA system board



[HP Z820 mainboard from Wikimedia by Jud McCranie CC BY-SA 4.0]

On-chip communication

- Use high-speed networks instead of conventional buses
  - Using ideas from computer networking
  - On-chip network can achieve high throughput and low latencies
- Example: on-chip ring network connecting 6 CPUs, a system controller ("agent") and a GPU

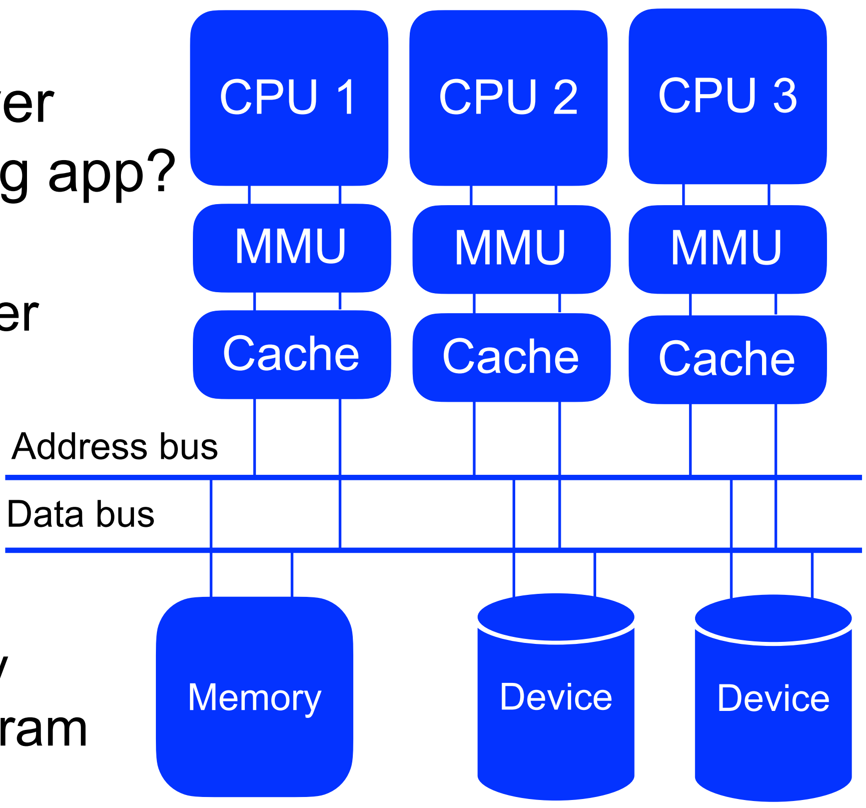


Heterogeneous systems: GPGPUs

- In modern computers, not only CPUs can execute code
- **GPGPU**s (general purpose graphics processing units)
  - Massively parallel processors for typical parallel tasks
  - 3D graphics, signal processing, machine learning, bitcoin mining…
  - Few features for protection, security…
- Traditionally, GPUs were accessible to a single program only (in Unix: “X window server”) for drawing
  - Other programs had to ask the X server for services
- In modern systems, multiple programs want direct access to the GPGPU
  - How can the OS multiplex the GPGPU *safely* and *securely?*

Security

- …there’s another important non-functional property!
- Multiple programs running simultaneously
  - e.g. a online banking application and a video player
- How can be avoid the video player accessing memory of the banking app?
  - e.g. your account number and password, which the video player could share online!
- Restrict access to non permitted memory ranges
  - The memory management unit (MMU) only makes memory ranges visible to a running program “belonging” to it



The MMU

- Idea: **intercept** “virtual” addresses generated by the CPU
  - MMU checks for “allowed” addresses
  - It translates allowed addresses to “physical” addresses in main memory using a **translation table**

- Problem: translation table for each single address would be large
  - Split memory into **pages** of identical size (power of 2)
  - Apply the same translation to all addresses in the page: **page table**

- MMUs were originally separate ICs sitting between CPU and RAM
  - Or even realised using discrete components (e.g. in the Sun 1 [8])
  - Higher integration due to Moore’s Law ➛ fit on CPU chip now!

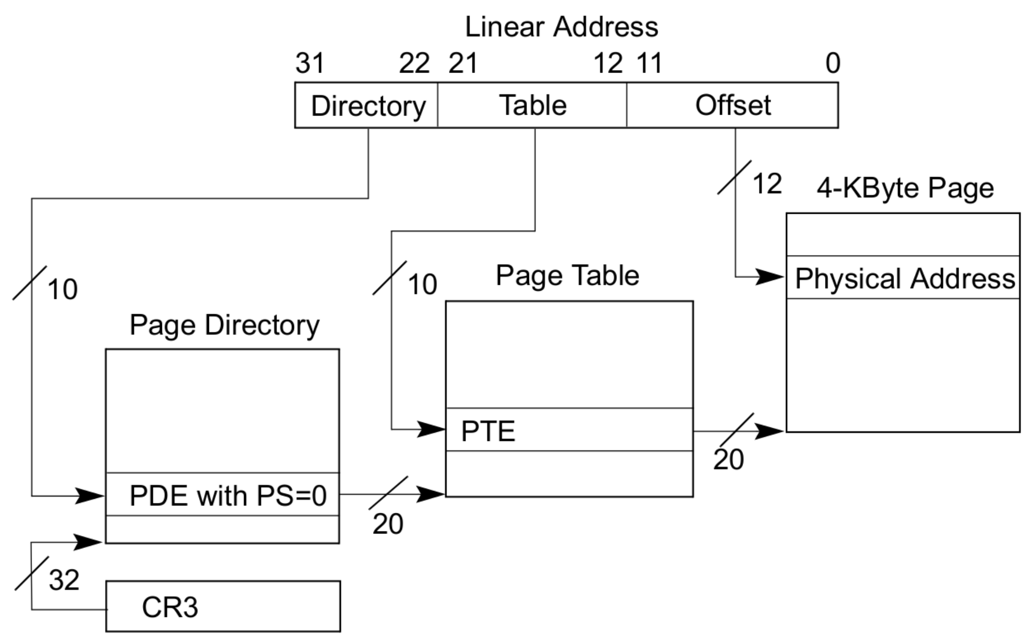
Motorola 68451 MMU chip (1982) [7]

Page table structure

- Split memory into **pages** of identical size (power of 2)
- Apply the same translation to all addresses in the page: **page table**
- Find a compromise **page size** allowing flexibility and efficiency
  - Typically several kB: 4 kB=2^{12} bytes (x86), 16 kB (Apple M1)
- 32 bit CPU (2^{32} addr.): 4 kB pages ➛ $2^{32}/2^{12} = 2^{20}$ pages ~ 1 million!
  - Use **sparse multi-level** page tables ➛ reduce page table size

For 32 bit x86:

- Page size:
  - 2^{12} = 4096 bytes
- Page table:
  - 2^{10} page entries
- Page directory:
  - 2^{10} page tables

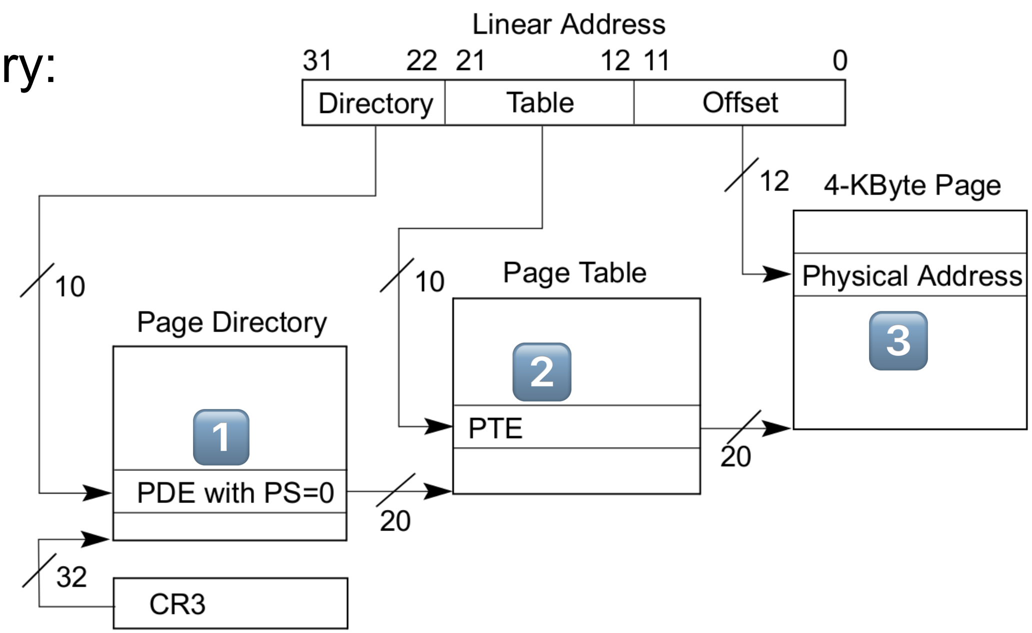


The memory translation process

- The MMU splits the virtual (or “linear”) address coming from the CPU into three parts:
  - 10 bits (31–22) page directory entry (PDE) number
  - 10 bits (21–12) page directory entry (PTE) number
  - 12 bits (11–0) page offset inside the references page (untranslated)

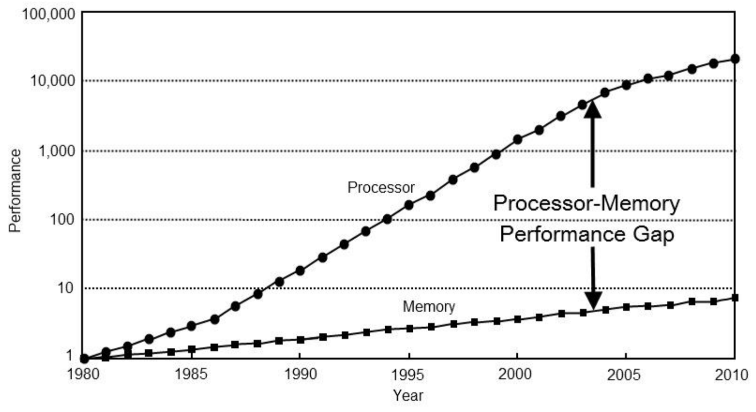
Translation process:

1. Read PDE entry from directory:
   ➛ address of one page table
2. Read PTE entry from table:
   ➛ physical base address of memory page
3. Add offset from original virtual address (bits 11–0) to obtain the complete physical memory address



Speeding up translation

- Where is the page table stored?
  - Can be several MB in size
    ➛ doesn't fit on the CPU chip!
  - Page directory and page tables are **in main memory!**



- Using virtual memory address translation requires **three main memory accesses!**
  - Same idea as with regular slow memory access: use **cache!**
- The MMU uses a special cache on the CPU chip:
  the **Translation Lookaside Buffer** (TLB)
  - Caches commonly (most often? most recently?) used PTEs
  - The locality principle at work again
- More details on this an upcoming lecture…

What about the operating system?

- New hardware capabilities have to be used efficiently
- The operating system has to manage and multiplex the related resources
  - ➛ The **OS has to adapt** to new hardware capabilities!
  - ➛ It has to provide **code** for all new capabilities
  - ➛ These often **interact** with other parts of the system, making the overall OS more complex
- A modern OS also has to ensure adherence to non-functional requirements (security, energy, real-time, …)
  - The OS has to do more bookkeeping and statistics
  - Some of the non-functional properties contradict each other
  - Unexpected problems may show up (Meltdown, Spectre [5,6])
- Finally, the OS itself has to be efficient!

References

1. John von Neumann, First Draft of a Report on the EDVAC, 1945 – reproduced in IEEE Annals of the History of Computing, vol. 15, no. 4, pp. 27-75, 1993
2. U. Drepper, What Every Programmer Should Know About Memory, RedHat Inc., 2007
3. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel, Scratchpad memory: design alternative for cache on-chip memory in embedded systems, Proceedings of the tenth international symposium on hardware/software codesign, 2002
4. Gordon E. Moore, Gordon E., Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff., in IEEE Solid-State Circuits Society Newsletter, vol. 11, no. 3, pp. 33-35, Sept. 2006
5. Moritz Lipp et al., Meltdown: Reading Kernel Memory from User Space, 27th USENIX Security Symposium 2018
6. Paul Kocher et al., Spectre Attacks: Exploiting Speculative Execution, 40th IEEE Symposium on Security and Privacy 2019
7. Motorola Semiconductors Inc., MC68451 Memory Management Unit, document nr. ADI-872-R1, 1983
8. Sun Microsystems Inc., Sun-1 System Reference Manual, P/N 800-0345, 1982