

# CPSC 457

Hardware, booting, cache, kernel

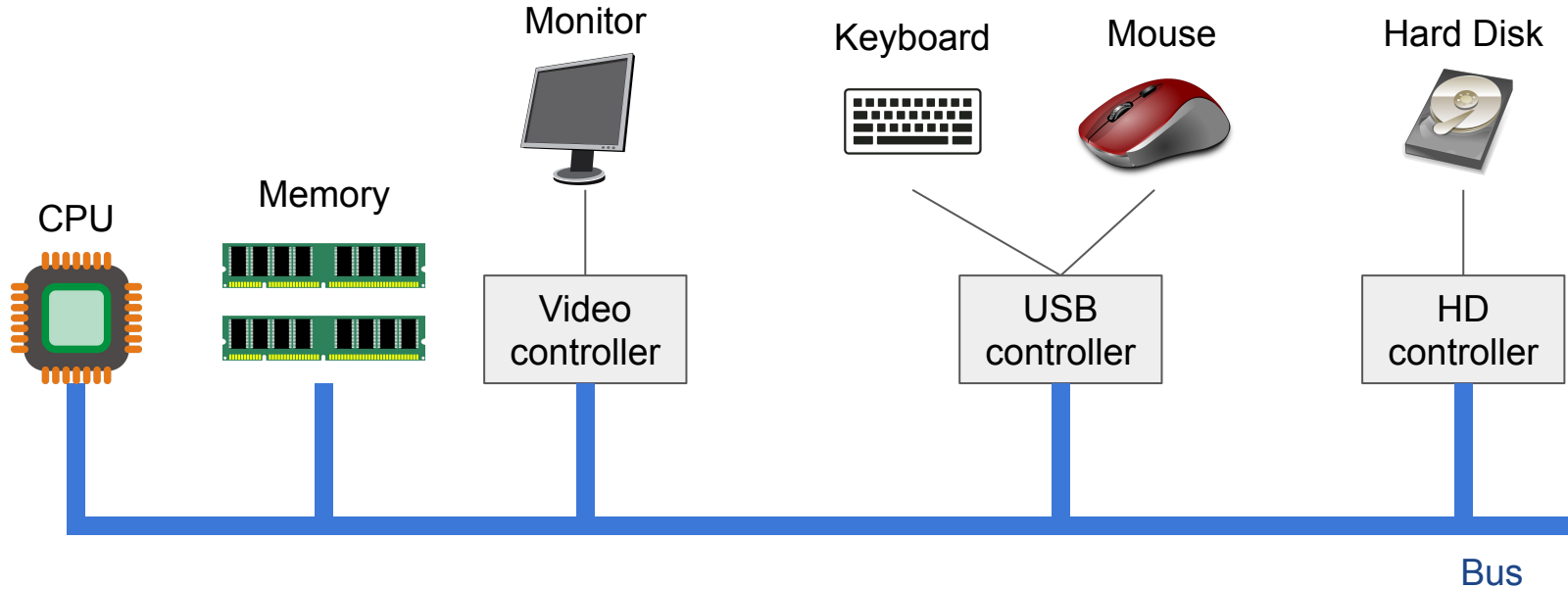
Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

# Outline

---

- hardware review
- caching
- booting
- kernel mode, user mode

# Hardware review



common components of a desktop computer

- the “brain” of the computer
- on-board registers for faster computation
  - instead of accessing memory for every instruction
  - accessing information in registers is much faster than memory
  - general purpose registers:
    - data & addresses
  - special purpose registers:
    - program counter: contains memory address of the next instruction to be fetched
    - stack pointer: points to the top of the current stack in memory
    - status register: interrupt flag, privilege mode, zero flag, carry flag, ...
  - other (floating point, vector, internal, machine specific, etc)

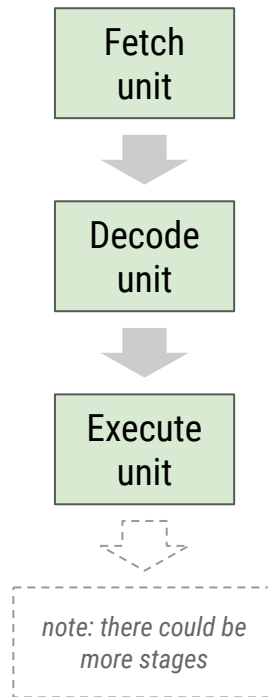


# Instruction cycle

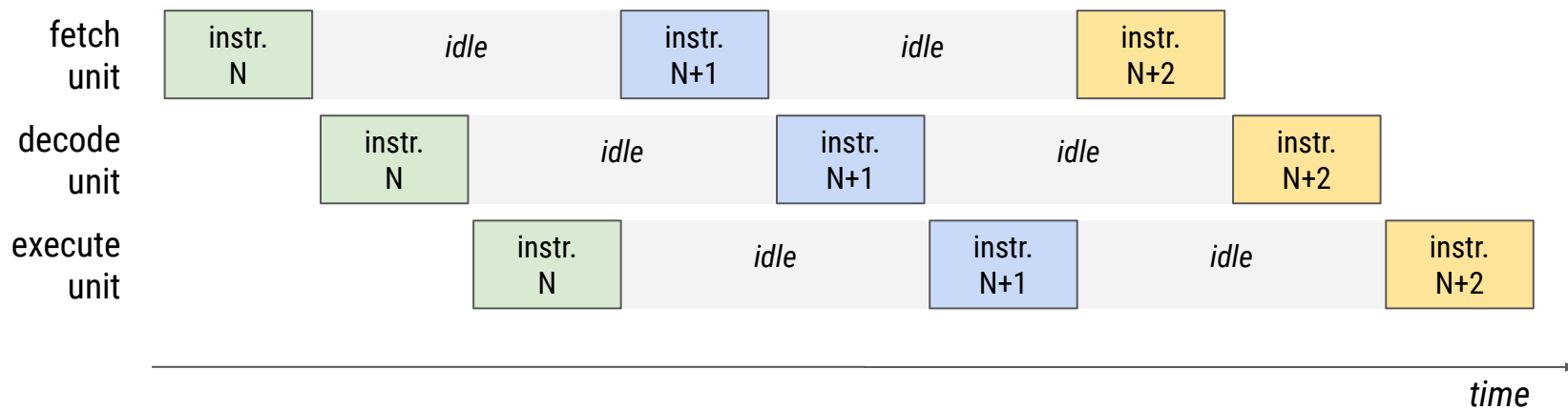
- a simple CPU cycle when stages operate sequentially:

1. **fetch** an instruction from memory
2. **decode** it to determine its type and operands
3. **execute** it
4. repeat

- fetch from memory is usually the longest operation



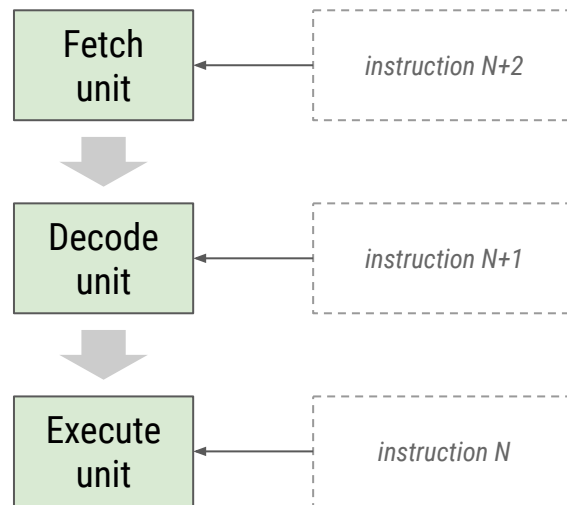
# Instruction cycle - sequential



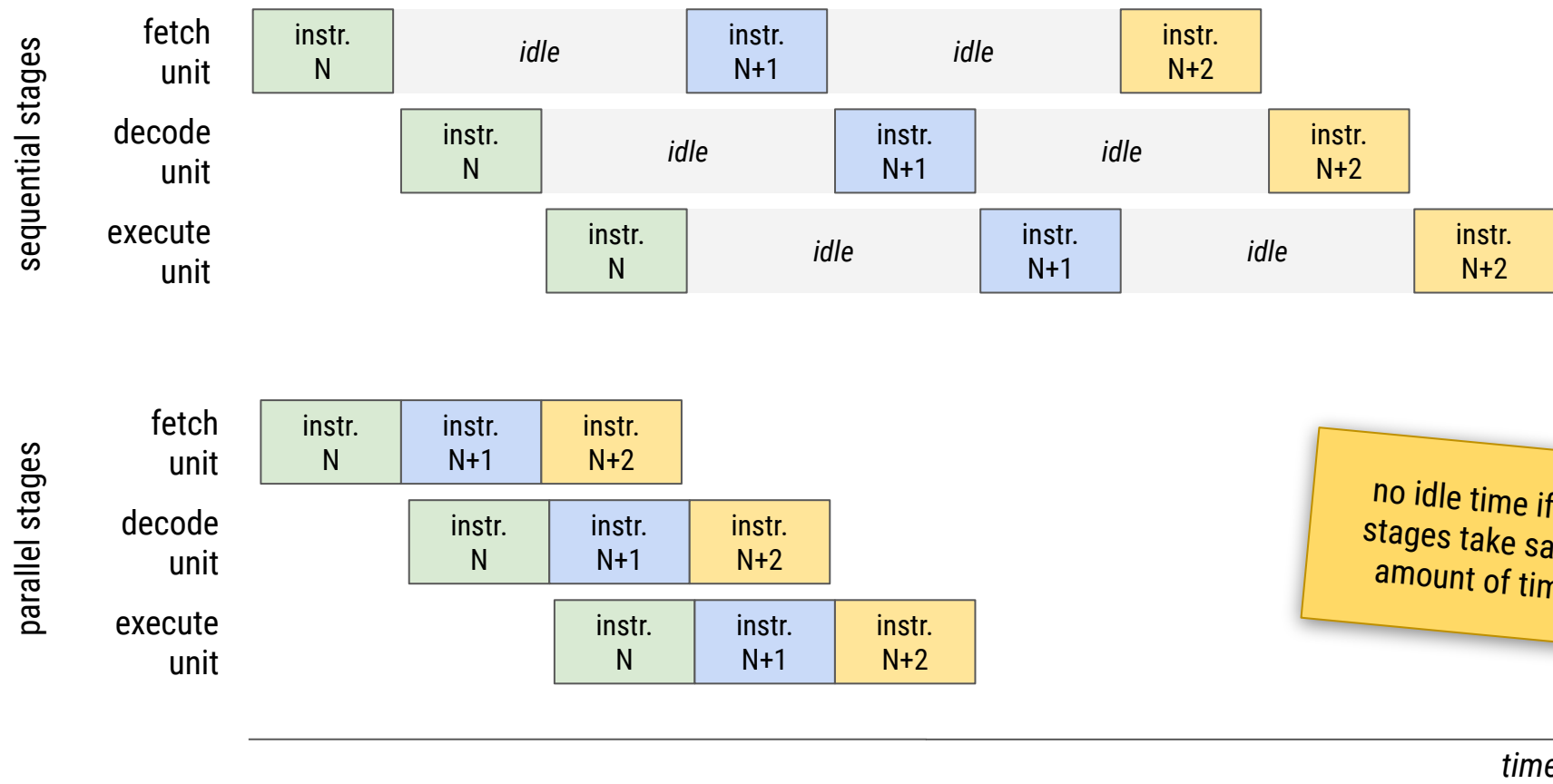
- assuming all stages take the same amount of time
- notice that the units are often idle
- we can improve the performance by letting the units operate in parallel

# Instruction pipelining

- we can let the stages work in parallel
- while executing instruction N, the CPU could be decoding instr. N+1 and fetching instr. N+2



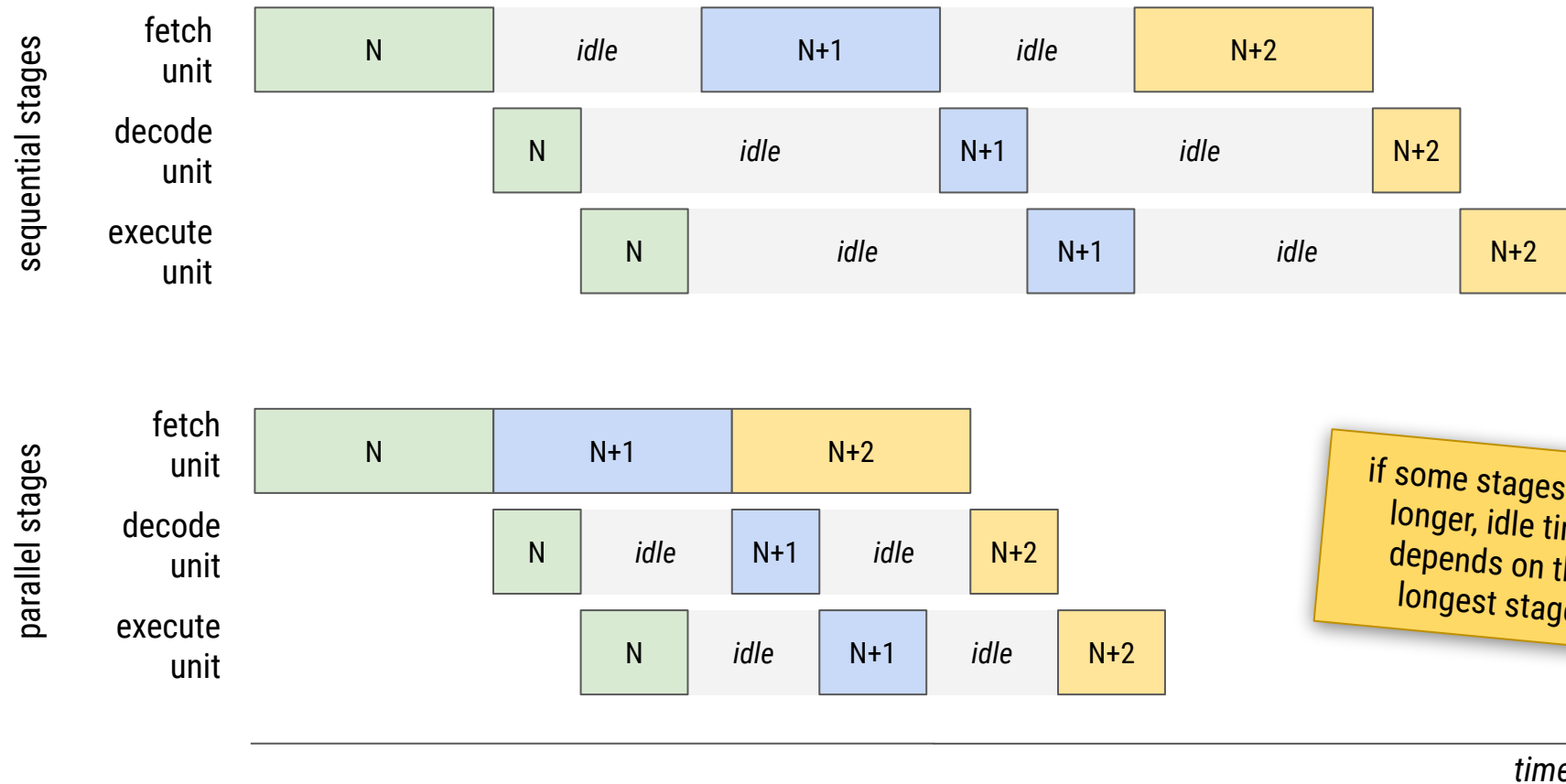
# Instruction cycle – sequential vs parallel



no idle time if all stages take same amount of time



# Instruction cycle – sequential vs parallel



# Instruction pipelining

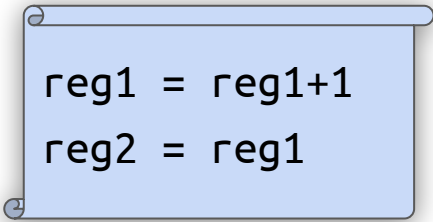
---

- benefits:

- the CPU can execute more than one instruction at a time
- this allows the CPU to mask some of the memory access time

- cons:

- more complexity
- have to deal with invalidated stages



```
reg1 = reg1+1  
reg2 = reg1
```

# Instruction pipeline – example 1

---

- consider a CPU with a 3 stage instruction pipeline, where each stage takes  $1/1000\text{s}$
- how many instructions per second can this CPU execute?
- if the stages are executed sequentially?  
every instruction takes  $0.003\text{s}$ , so on **average** the CPU executes  $\sim 333$  instructions/s
- if the stages are executed in parallel?  
first instruction will take  $0.003\text{s}$  to execute...  
but over **long run** the CPU will execute  $1000$  instructions/s

# Instruction pipeline – example 2

- consider a CPU with a 3 stage instruction pipeline,  
fetch takes 10ns, decode takes 3ns and execute takes 2ns
- how many instructions per second can this CPU execute?

- if the stages are executed sequentially?

each instruction takes  $10\text{ns} + 3\text{ns} + 2\text{ns} = 15\text{ns}$

on average the CPU executes  $1 \text{ instr.} / 15\text{ns} \rightarrow \sim 66,666,667 \text{ instructions/s}$

- if the stages are executed in parallel?

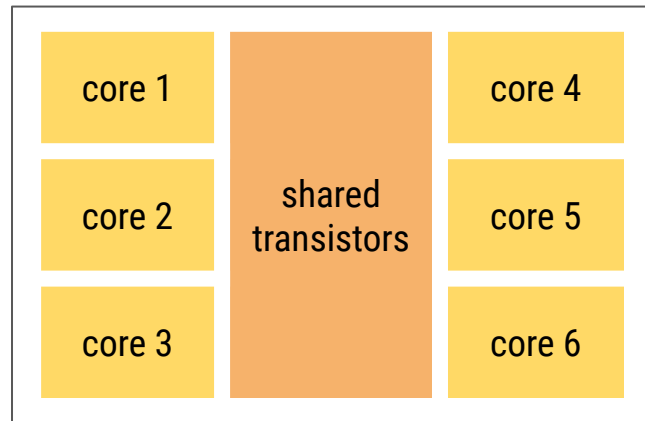
first instruction will take  $10\text{ns} + 3\text{ns} + 2\text{ns} = 15\text{ns}$  to execute

but over long run the CPU will execute  $1 \text{ instr.} / 10\text{ns} = 100,000,000 \text{ instructions/s}$

the pipeline is as slow as its slowest stage

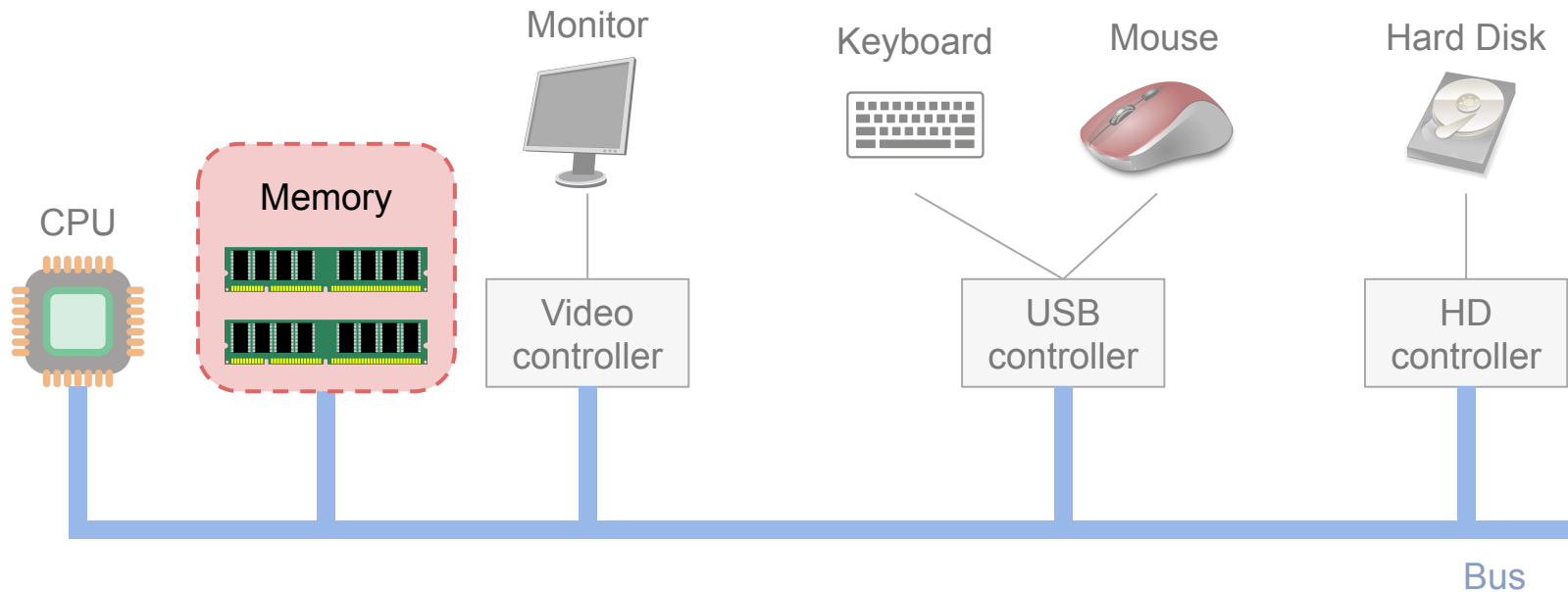
# Multicore CPUs

- how to make CPUs run our programs faster?
  - option 1 - run each instruction faster (more GHz, faster memory, ...)
  - option 2 - run multiple instructions simultaneously (more transistors)
  - option 3 - both of the above
- nearly all modern CPUs contain multiple cores
- a core = "mini CPU"
- each core can execute code in parallel with other cores
- cores can share some hardware, eg. cache(s)

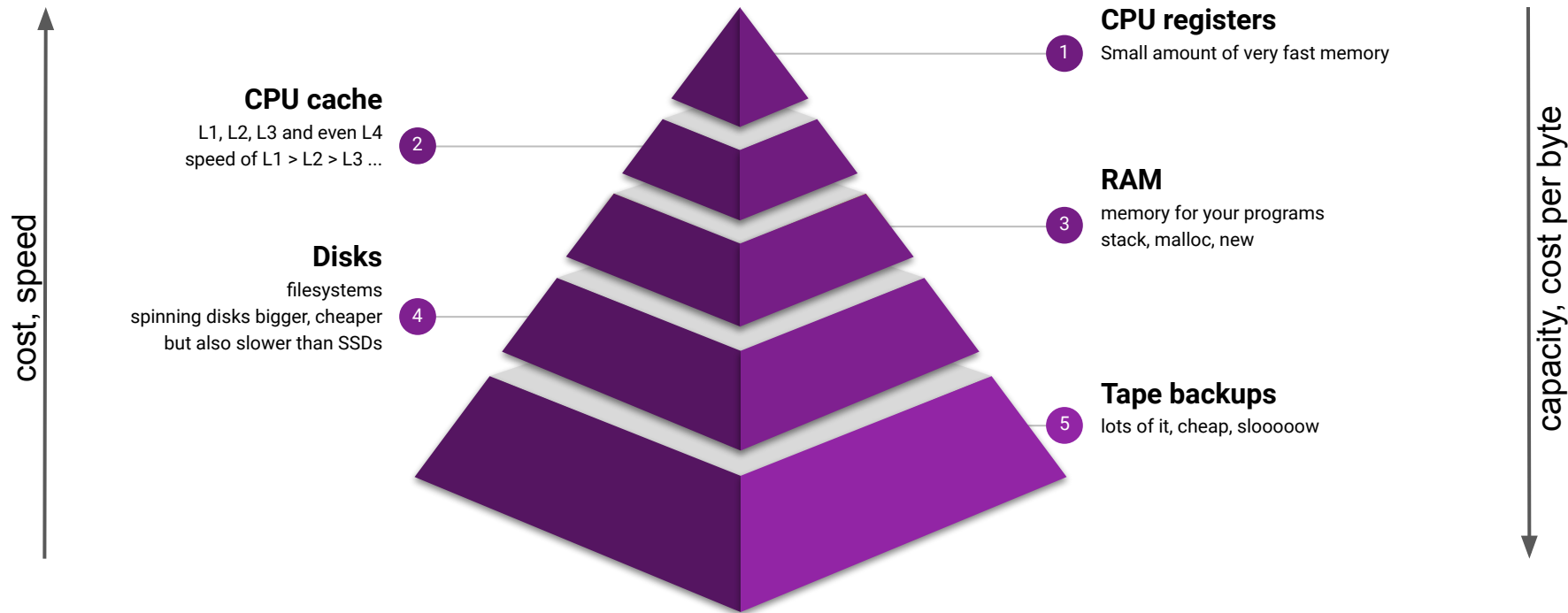


# Memory

- ideally, memory should be (i) fast, (ii) large and (iii) cheap
- in practice, we can get 2 of the 3, but not all three



# Typical memory hierarchy



# Memory

- main memory: random-access memory (RAM)
- consists of an array of words, and each word has its own address (memory address)
- typical memory operations:
  - `load <address>,<register>` – load a word from memory into CPU register
  - `store <register>,<address>` – stores contents of register in memory
- both are slow operations compared to the speed of the CPU

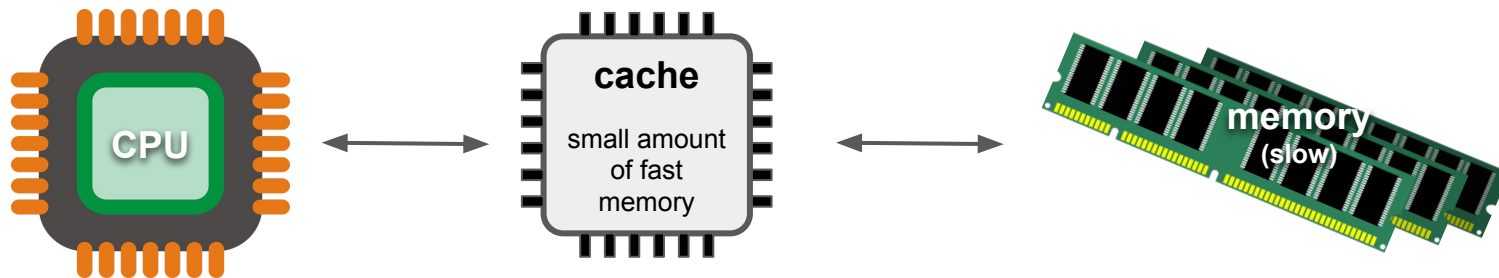




# Caching

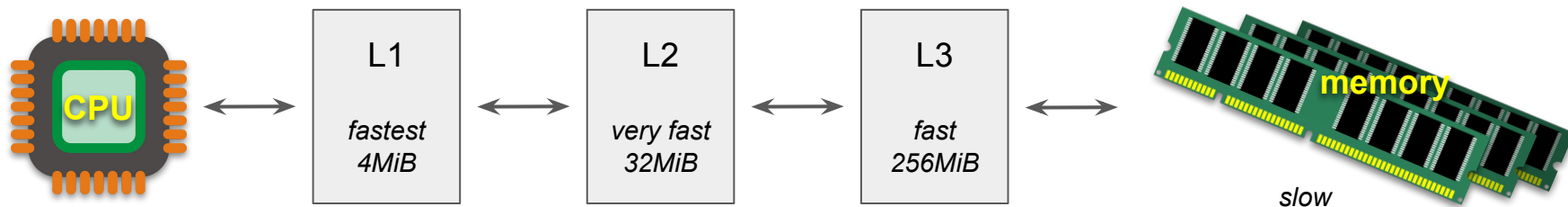
## ■ CPU caching

- most heavily used data from memory is kept in a high-speed cache located inside or very close to the CPU
- when CPU needs to get data from memory, it first checks the cache
- **cache hit**: the data needed by the CPU is in the cache
- **cache miss**: CPU needs to fetch the data from main memory



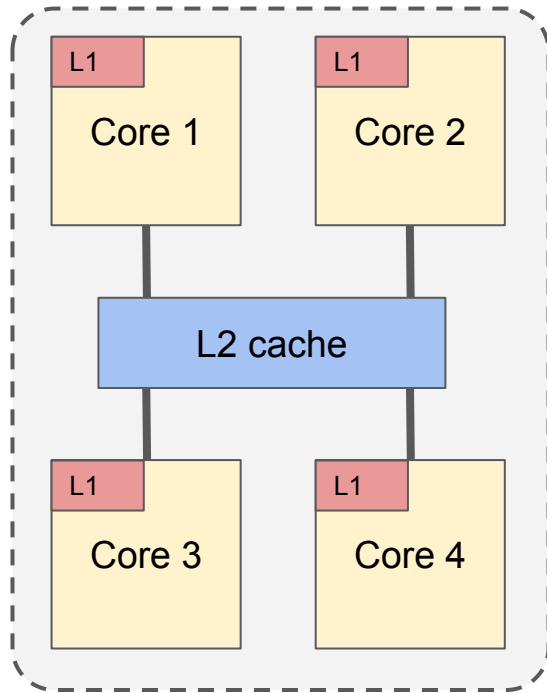
# Cache hierarchy (multilevel caches)

- L1 cache (~32KiB): fastest, feeds decoded instructions into CPU execution engine, private per core
- L2 cache (½ MiB): stores recently used memory, slower than L1, may be shared by multiple cores
- L3 (x MiB): faster than memory, slower than L2, usually shared by all cores, or a group of cores
- L1, L2 and L3 are usually on the same chip as the CPU
- some CPUs have L4 cache ...

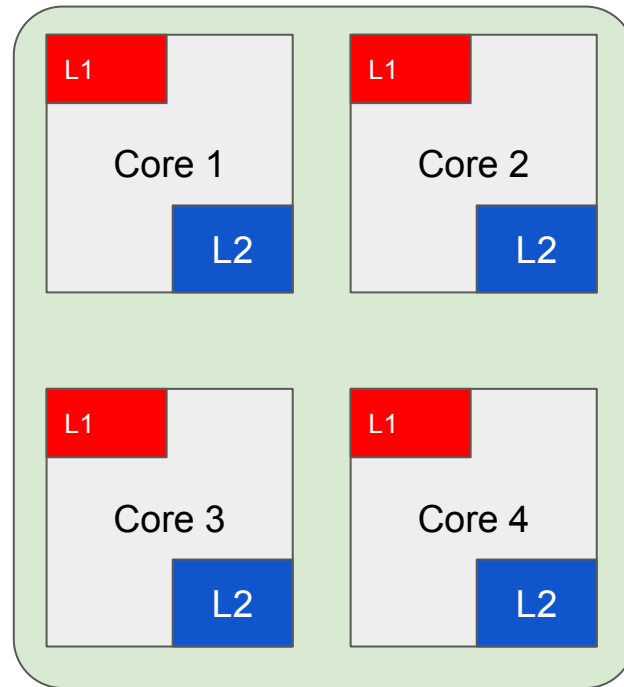


AMD Ryzen™  
Threadripper™  
3990X

# Caches on multicore CPUs



(a) A quad-core chip with a shared L2 cache.



(b) A quad-core chip with separate L2 caches.

# CPU cache and C++

- let's time a simple C++ loop:

```
auto start_time = clk::now(); // start the timer
int sum = 0;
```

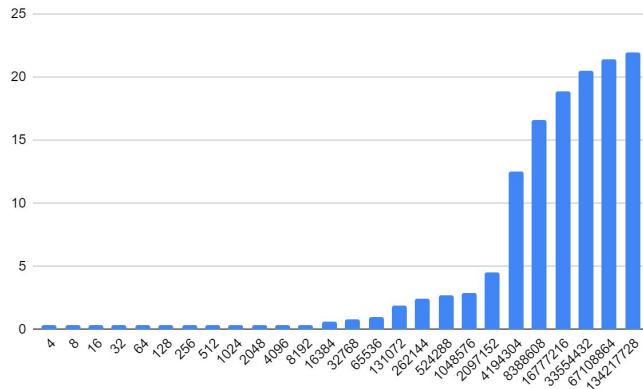
```
for( int i = 0 ; i < N ; i ++ ) {
    sum += arr[arr[i % msize]];
}
```

```
auto end_time = clk::now(); // end the timer
```

```
// report results
```

```
double dt = duration_cast<duration<double>>(end_time - start_time).count();
printf("%9.2lfK %10.4lf /*%d*/\n", msize * sizeof(int) / 1024.0, dt, sum);
```

- notice the body of the loop executes the same number of times
- but depending on the size of the array `msize`, and the contents of the array, the loop can execute very fast or very slow
- <https://repl.it/@pfederl/cpu-cache-test>



- the goal of caching is to increase performance of slower memory/device by adding a small amount of fast memory (cache)
- improving read performance:
  - keep copy of information obtained from slow storage in cache
  - next time we need the information, check the cache first
- improving write performance
  - write info to fast storage, and eventually write to slow storage
- caching is a very useful concept in general
- many uses: disk cache, DNS, database
- cache storage is fast but expensive, so it's usually much smaller than the slow storage

# Caching

---

- some general caching issues:
  - when to put a new item into the cache
  - which cache line to put the new item in
  - which item to remove from the cache when cache is full
  - where to put a newly evicted item in the larger memory
  - multiple cache synchronization
  - how long is the data in cache valid (expiration)
- answers depend on the application

# Memoization

---

- similar concept to caching
- optimization technique used to speed up programs, by storing results of expensive computations

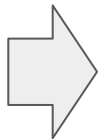
```
def fib_slow(n):  
    if n < 2:  
        res = n  
    else:  
        res = fib_slow(n-1)  
              + fib_slow(n-2)  
    return res
```

# Memoization

- similar concept to caching
- optimization technique used to speed up programs, by storing results of expensive computations

```
def fib_slow(n):  
  
    if n < 2:  
        res = n  
    else:  
        res = fib_slow(n-1)  
              + fib_slow(n-2)  
    return res
```

$O(2^n)$



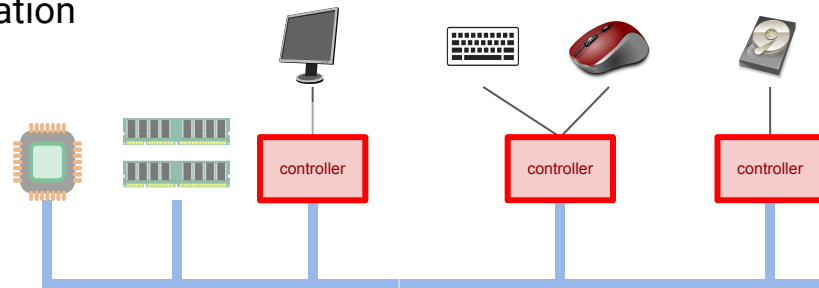
```
cache = {}  
def fib_fast(n):  
    if n not in cache.keys():  
        if n < 2:  
            cache[n] = n  
        else:  
            cache[n] = fib_fast(n-1)  
                      + fib_fast(n-2)  
    return cache[n]
```

$O(n)$



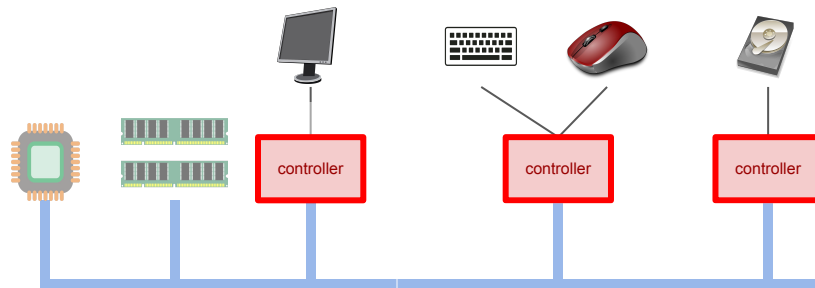
# Hardware review - I/O

- I/O devices usually connected to computer via device controller
- **device controller**
  - a chip or a set of chips that physically control the device
  - controlling the device is complicated, and CPU could be doing other things, so the controller presents a simpler interface to the OS
  - there are many different types of controllers
- **device**
  - connects to the computer through the controller
  - follows some agreed standard for communication

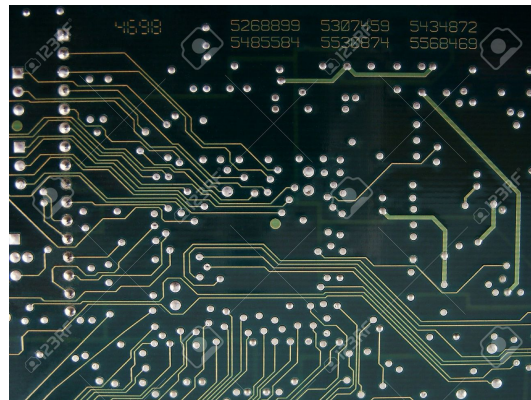
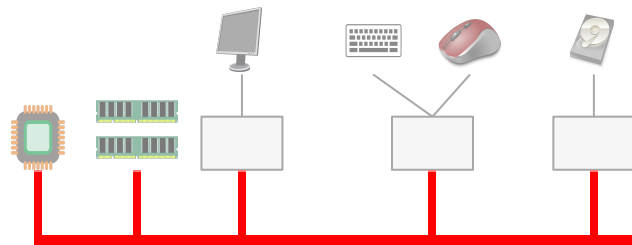


## ■ device driver

- is the software which OS uses to talk to a device
- usually written by the controller manufacturer, following some abstraction defined by OS
- needed so that an OS knows how to communicate with a device (or controller)
- drivers are often implemented as kernel modules, loaded on demand, running in kernel mode



- a communication system for transferring data between different computer components
- modern computer systems have multiple busses, eg. cache, memory, PCI, ISA, etc
- each has a different transfer rate and function
- OS must be aware of all of them for configuration and management
- for example, collecting information about the I/O devices
- assigning interrupt levels and I/O addresses
- much of this is done during the boot process



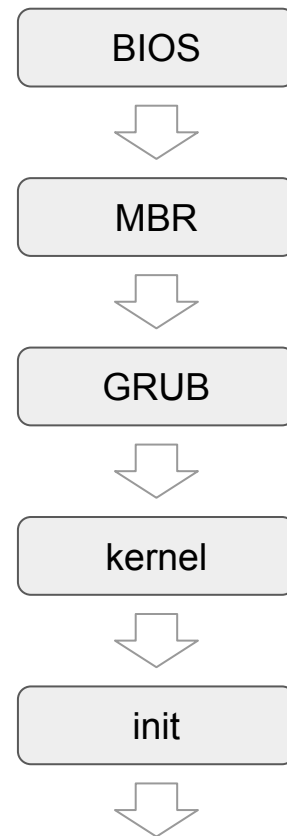
# Booting



<http://www.youtube.com/watch?v=C2Ph8zwpNyl&feature=related>

# Booting a (Linux) system

- when the computer is booted, the BIOS is started  
(Basic Input Output System) is a program stored on motherboard
- check the RAM, keyboard, other devices by scanning the ISA and PCI buses
- record interrupt levels and I/O addresses of devices, or configure new ones
- determine the boot device (ie. try list of devices stored in CMOS)
- read & run primary boot loader program from first sector of boot device
- read & run secondary boot loader from potentially another device
- read in the OS from the active partition and start it
- OS queries the BIOS to get the configuration information and initialize all device drivers in the kernel
- OS creates a device table, and necessary background processes, then waits for I/O events



# Kernel

---

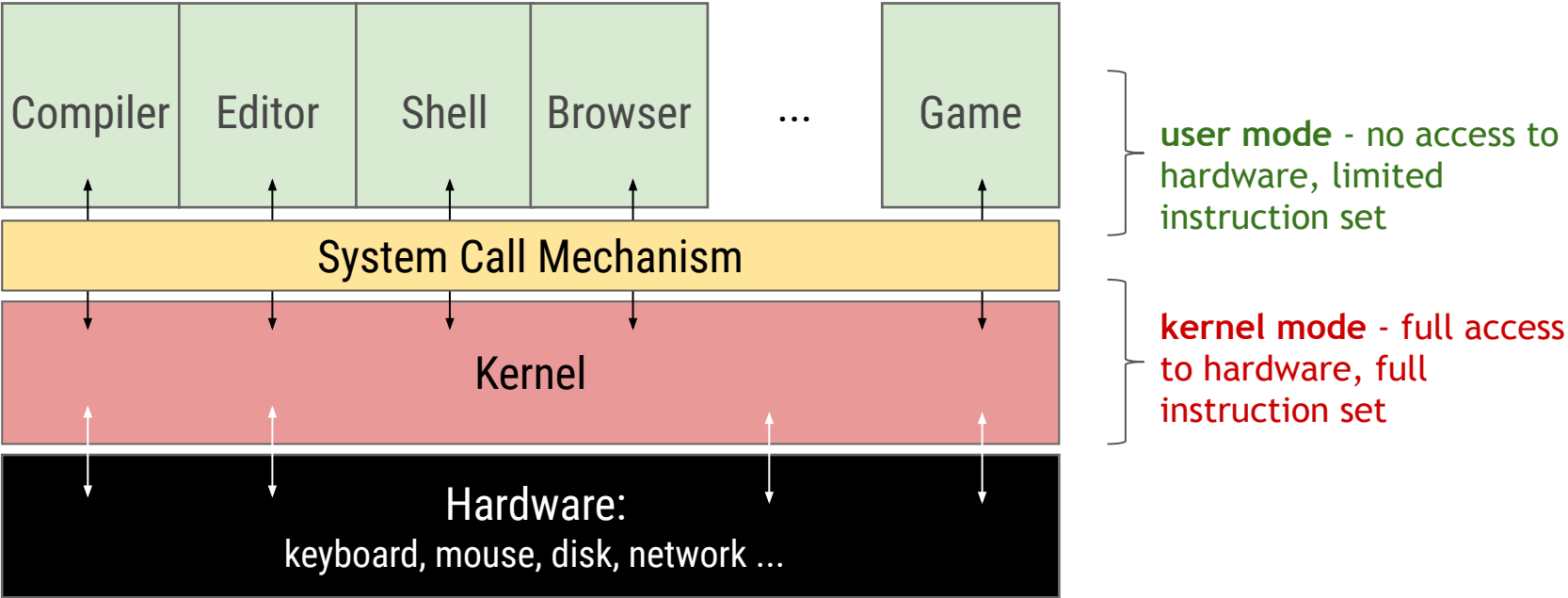
- the central part, or the "heart" of the OS
  - located and started by a bootstrap program (boot loader)
  - provides services to applications via system calls
  - handles all interrupts
  - most of the kernel is a set of routines, some invoked in response to interrupts, traps, etc.
  - kernel is "running" at all times on the computer

# Kernel mode

---

- most modern CPUs support at least two privilege levels: kernel mode and user mode
- the mode is usually controlled by modifying the status register
- when CPU is in **kernel mode** (aka unrestricted, privileged, supervisor mode):
  - all instructions are allowed
  - all I/O operations are allowed
  - all memory can be accessed
  - note: *most* of the kernel runs in kernel mode, and only kernel runs in kernel mode
- when OS runs a normal application, it runs it in a **user mode**:
  - only some operations are allowed, the rest are disallowed
  - eg. accessing the status register is disallowed – of course,  
I/O instructions not allowed, access to some parts of memory not allowed, ...
  - illegal instructions result in traps (exceptions)

# Kernel vs. user mode





# User mode

- all applications run in user mode, including ones that came with the OS
  - that means applications cannot talk to hardware... (directly)
  - so how do they read/write files?
- applications must ask the kernel to do I/O by invoking an appropriate **system call**
  - system call = calling a kernel routine
  - cannot be a simple function call – why not?
  - we need a mechanism to safely switch from user mode to kernel mode
- **trap**
  - often a special instruction (**SWI** *n*, INT *n*, ... )
  - switches from user mode to kernel mode and invokes a predefined routine
  - think of it as 'pausing' the application and executing a kernel routine configured by the OS
  - when the kernel routine is done, user mode is restored and application 'resumes'
- a trap is a mechanism that OS can utilize to safely execute kernel routine in kernel mode

# Questions?

# Review

---

- Applications run in user mode.
  - **True** or False
- Device drivers run in kernel mode.
  - True or False or It Depends