UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

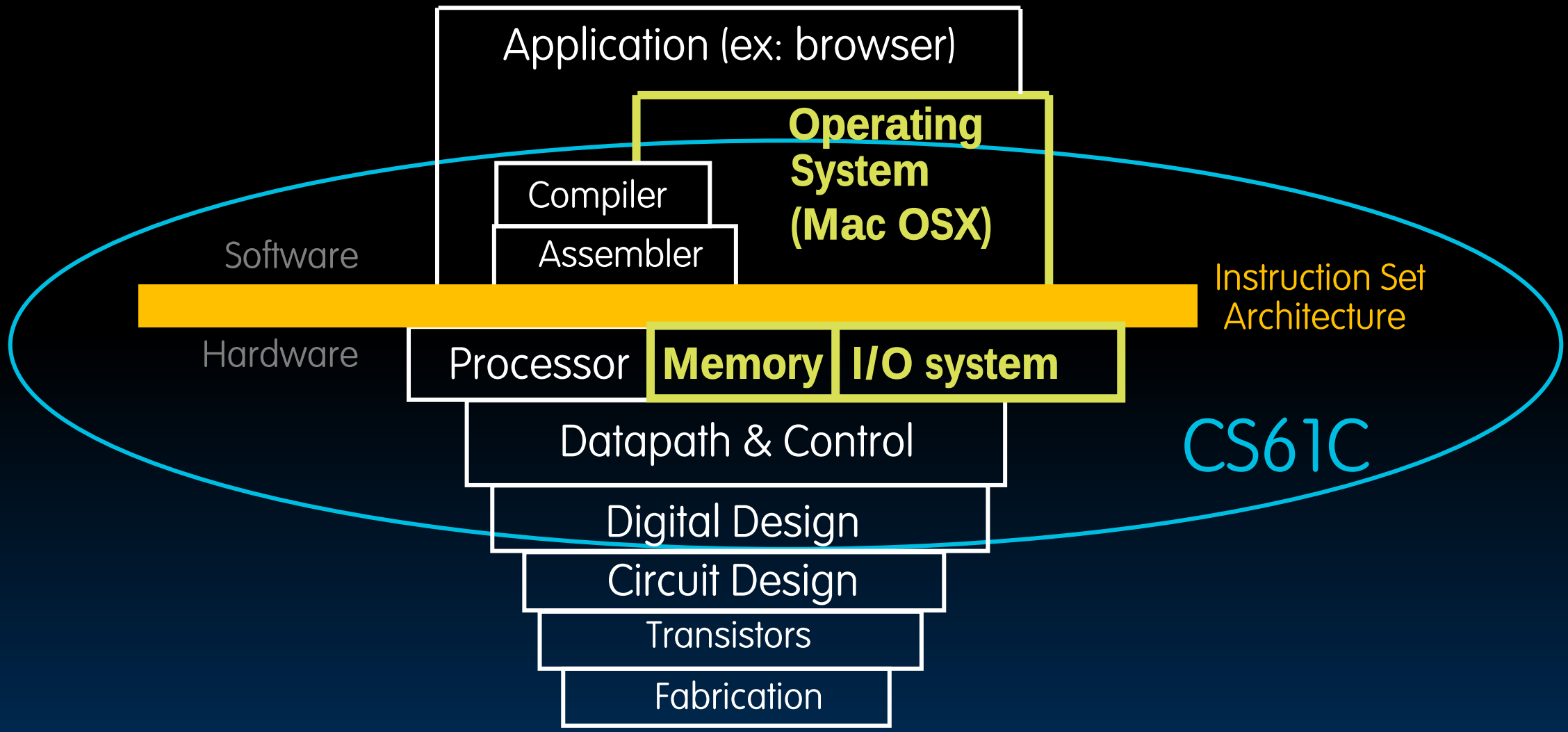
UC Berkeley
Lecturer
Justin Yokota

Virtual Memory I

The Computer

- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

Old-school Machine Structures



CS61C so far…

C Program/
Threaded C
Program

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

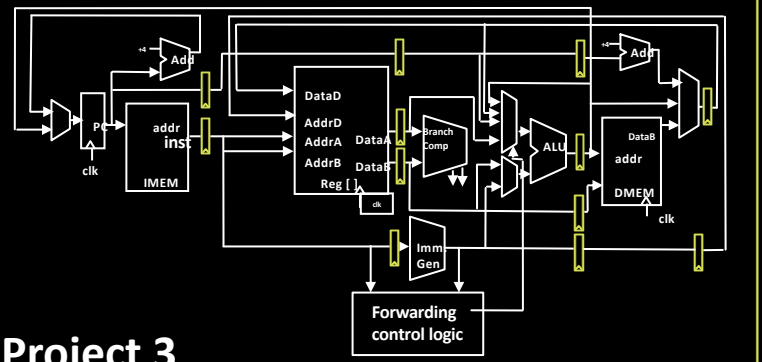
Project 1/
Project 4

RISC-V Assembly

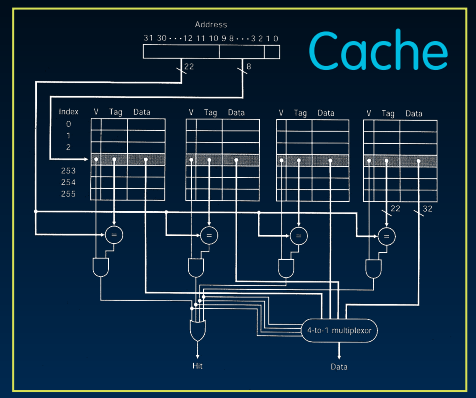
```
.foo
lw t0, 4(a0)
addi t1, t0, 3
beq t1, t2, foo
nop
```

Project 2

CPU



Project 3

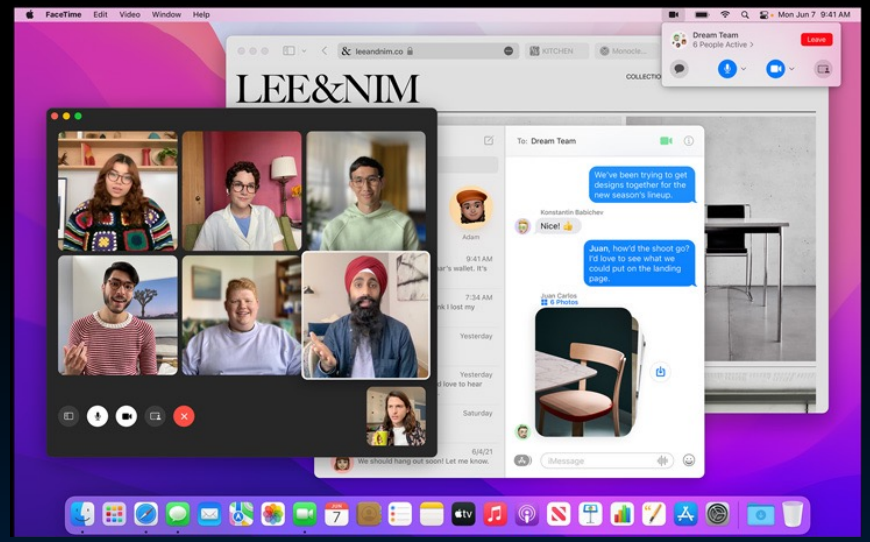


But Wait…

- When I run Venus, it only executes one program, then stops.
- My laptop looks and acts differently!

1. Multiple *I/O devices* (input-output)

when switched on…



2. Multiple programs running simultaneously "on" a software program called the *Operating System (OS)*

Raspberry Pi ($35) and I/O ports

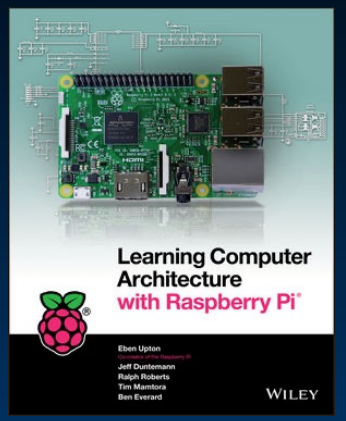
The Raspberry Pi is a low-cost computer.

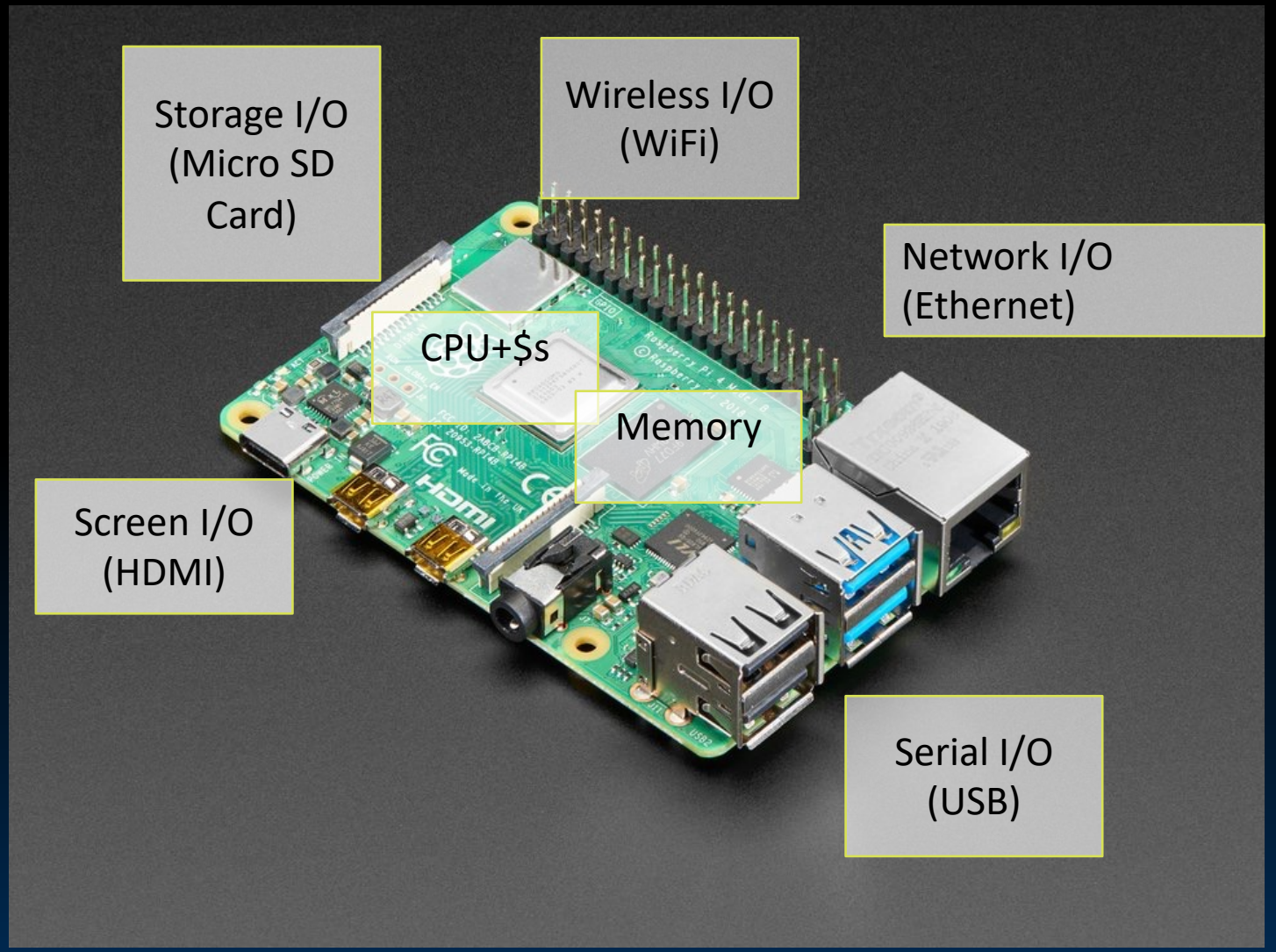
- Motherboard with CPU, I/O, caches, etc., soldered on
  - ("$" stands for cache)

CS61C w/Raspberry Pi?

- Free to download with UC Berkeley login!

https://onlinelibrary.wiley.com/doi/book/10.1002/9781119415534





The Next Step of CS61C

C Program/
Threaded C
Program

```
#include <stdlib.h>

int fib(int n) {
  return
    fib(n-1) +
    fib(n-2);
}
```

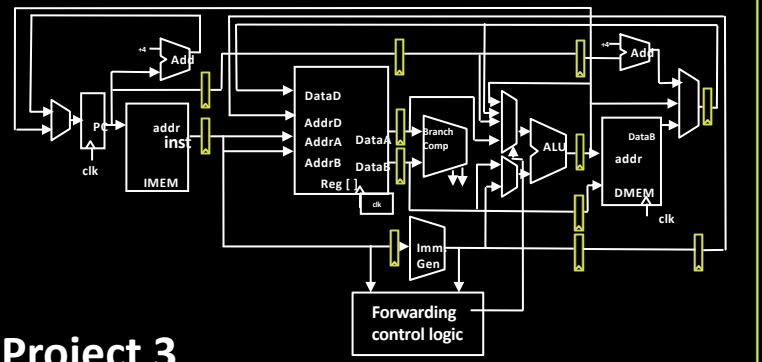
Project 1/
Project 4

RISC-V Assembly

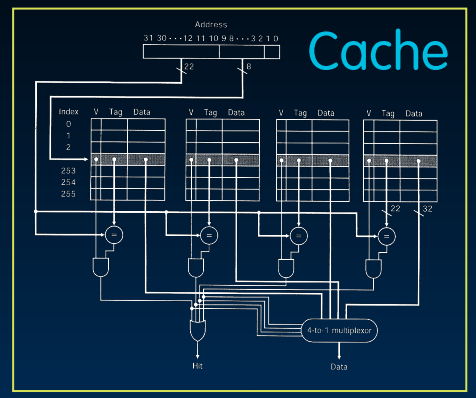
```
.foo
lw t0, 4(a0)
addi t1, t0, 3
beq t1, t2, foo
nop
```

Project 2

CPU



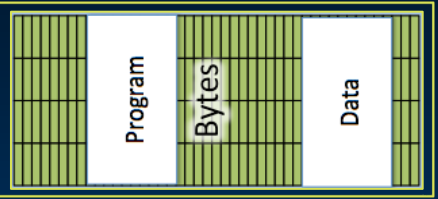
Project 3



Screen | Keyboard | Storage

I/O (Input/Output)

Memory



This module: Memory (DRAM), Storage (Disk), and I/O

OS Basics: Context Switching

- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

What does the core of the OS do?

- **OS is the (first) thing that runs when computer starts.**
  - Starts services (100+).
    - File system, Network stack (Ethernet, WiFi, Bluetooth, …), TTY (keyboard), etc.
- **Provides interaction with the outside world:**
  - Finds and controls all devices in the machine in a general way:
    - Relies on hardware specific "device drivers"
- **Loads, runs and manages programs:**
  - *Isolation*: Each program runs (i.e., appears to run) in its own little world.
  - Resource-sharing: Multiple programs share the same resources:
    - Memory
    - I/O devices: disk, keyboard, display, network, etc.
  - Time-sharing: Processor (CPU) runs multiple processes.

What does the core of the OS do?

- **OS is the (first) thing that runs when computer starts.**
  - Starts services (100+).
    - File system, Network stack (Ethernet, WiFi, Bluetooth, …), TTY (keyboard), etc.
- **Provides interaction with the outside world:**
  - Finds and controls all devices in the machine in a general way:
    - Relies on hardware specific "device drivers"
- **Loads, runs and manages programs:**
  - *Isolation*: Each program runs (i.e., appears to run) in its own little world.
  - Resource-sharing: Multiple programs share the same resources:
    - Memory
    - I/O devices: disk, keyboard, display, network, etc.
  - Time-sharing: Processor (CPU) runs multiple processes.

Over the next few lectures, we will discuss this entire slide. For now, we focus on resource-sharing *memory.*

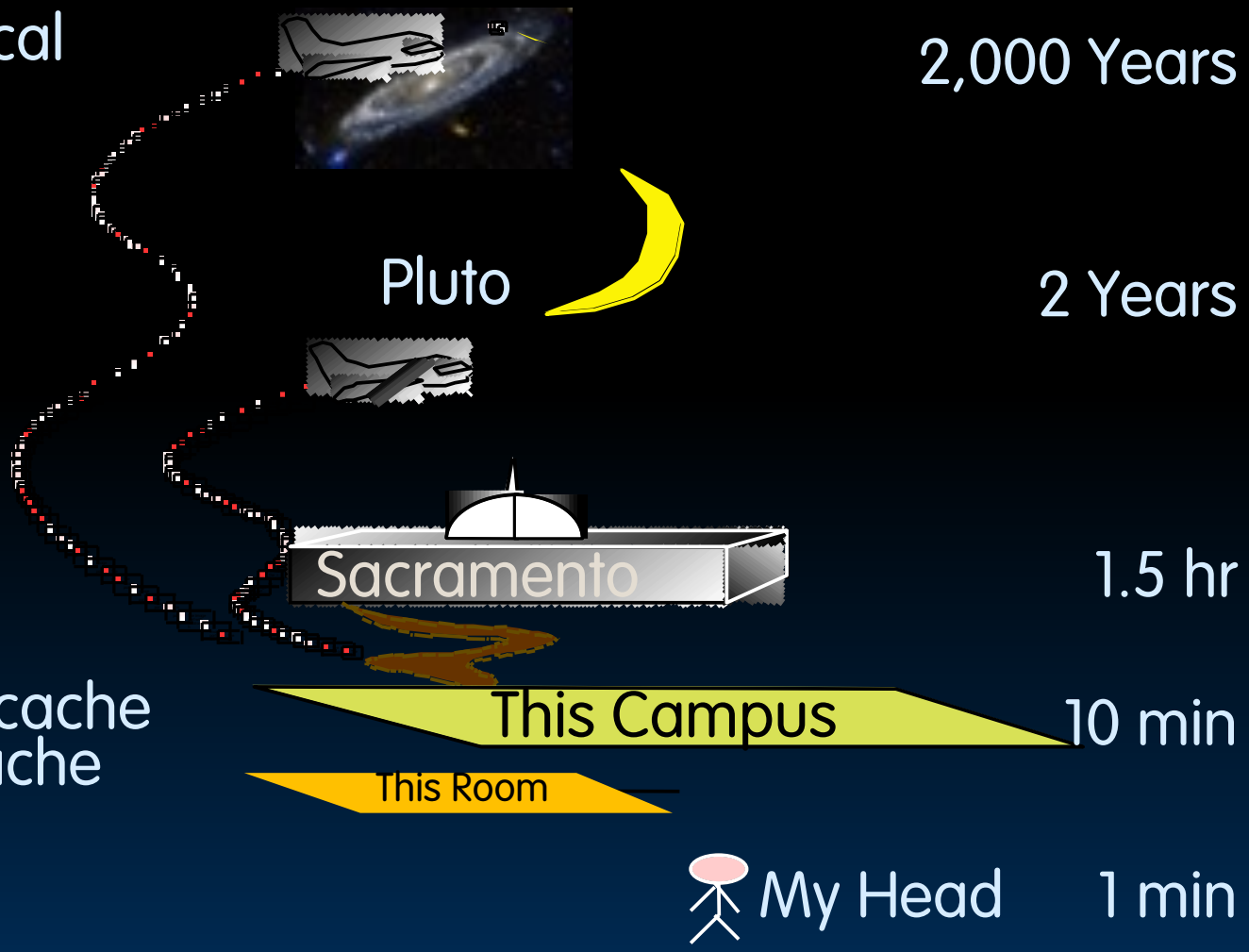
Multiprogramming at a High Level (more next time)

- **The OS manages multiprogramming, which is running multiple applications (processes) "simultaneously" on one CPU.**
  - (vs. multiprocessing: running processes simultaneously on different CPUs. The OS also manages this.)
- **This is achieved via OS context switches, i.e., switches between processes very quickly (on the human time scale):**
  - Save current process state (program counter, registers, etc.)
  - Load next process state to execute next instruction on CPU
  - Do *not* switch out data between main memory and disk! Too costly…

Storage Latency Analogy: How Far Away Is the Data?

Review

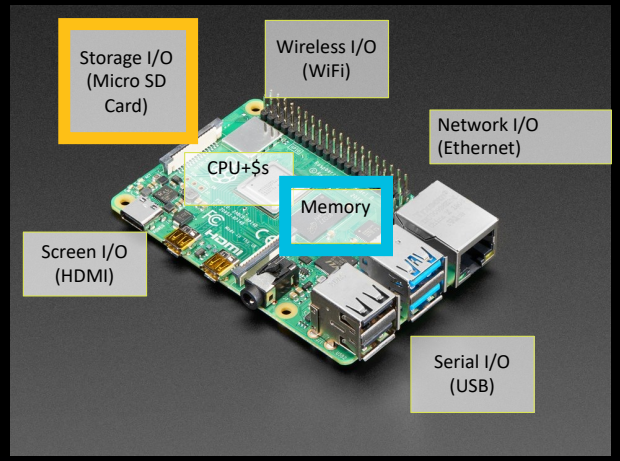
10^9	Tape /optical robot	2,000 Years
10^6	Disk	2 Years
100	Memory	1.5 hr
10	On-board cache	10 min
2	On-chip cache	
1	Registers	1 min
[ns]		



Physical Memory and Disk Storage

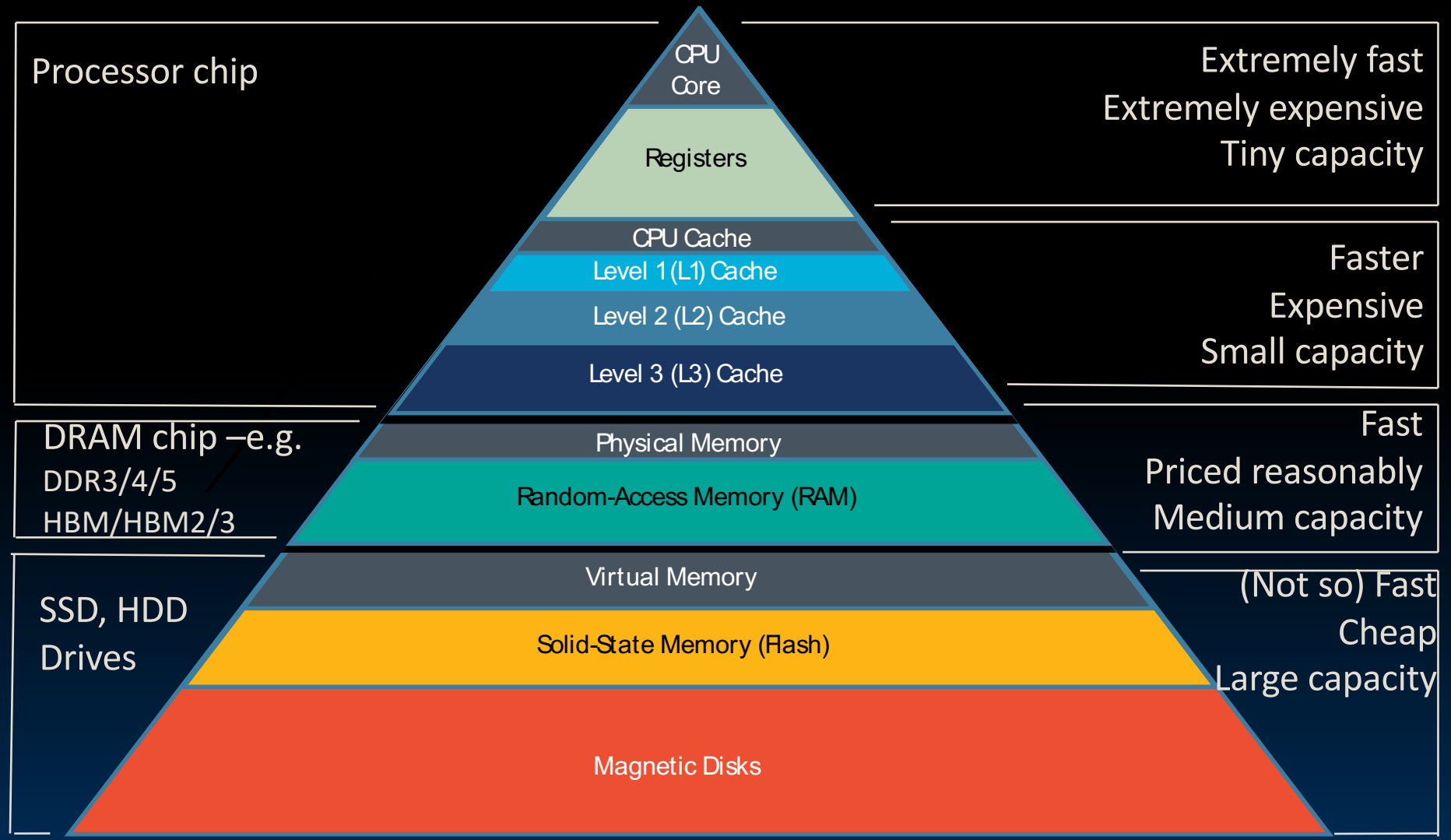
- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

Main Memory and Secondary Memory



Main memory

Secondary Memory



Main Memory is DRAM

- **Dynamic Random Access Memory:**
  - Latency to access first word: ~10ns (~30-40 processor cycles), each successive (0.5ns – 1ns)
  - Each access brings 64 bits, supports 'bursts'
  - $3/GiB
- **Data is impermanent:**
  - *Dynamic*: capacitors store bits, so needs periodic refresh to maintain charge
  - *Volatile*: when power is removed, loses data.
- **Contrast with SRAM (for caches):**
  - Static (no capacitors) but still volatile
  - Faster (0.5 ns)/more expensive/lower density

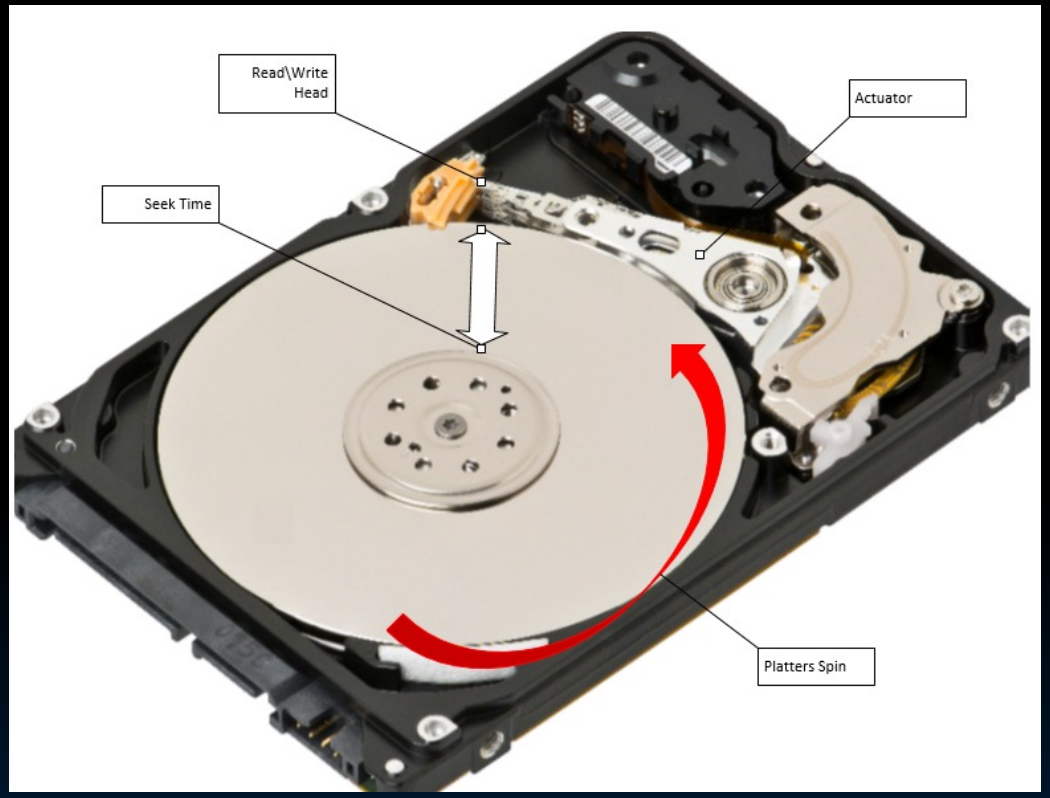
Desktop/server DIMM

Storage / "Disk" / Secondary Memory

- **Attached as a peripheral I/O device.** Non-volatile
- **Solid-State Drive (SSD)**
  - Access: 40-100µs (~100k proc. cycles)
  - $0.05-0.5/GB
  - Usually flash memory

- **Hard Disk Drive (HDD)**
  - Access: <5-10ms (10-20M proc. cycles)
  - $0.01-0.1/GB
  - Mechanical

Aside … How do HDDs work?

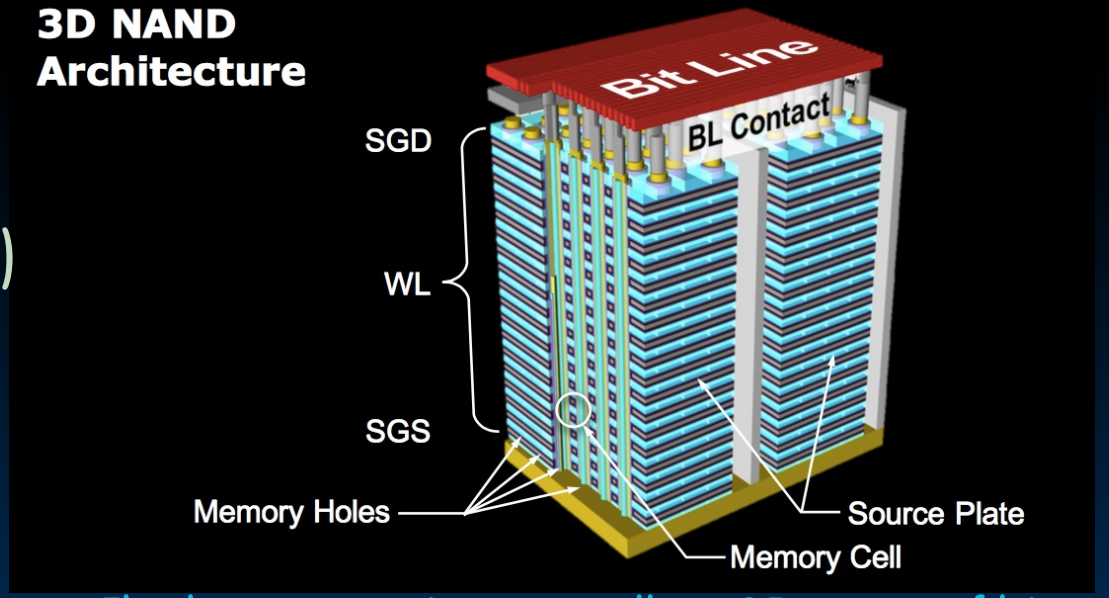


- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms (~10^7 processor cycles)

Nick Parlante's explanation:
https://cs.stanford.edu/people/nick/how-hard-drive-works/

Aside 2…What about SSDs?

- **Made with transistors (nothing mechanical/rotating)**
- **Operates like a "Ginormous" register file.**
  - Furthermore, does not "forget" when power is off (non-volatile)
- **Fast access to all locations, regardless of address**
  - Still much slower than register, DRAM.
  - Read/write blocks, not bytes.
  - Potential reliability issues… (read wear)
- **Some unusual requirements:**
  - Can't erase single bits – only entire blocks



Flash memory is generally a 3D array of bit cells (up to 256 layers!)

Virtual Memory and Virtual Addresses

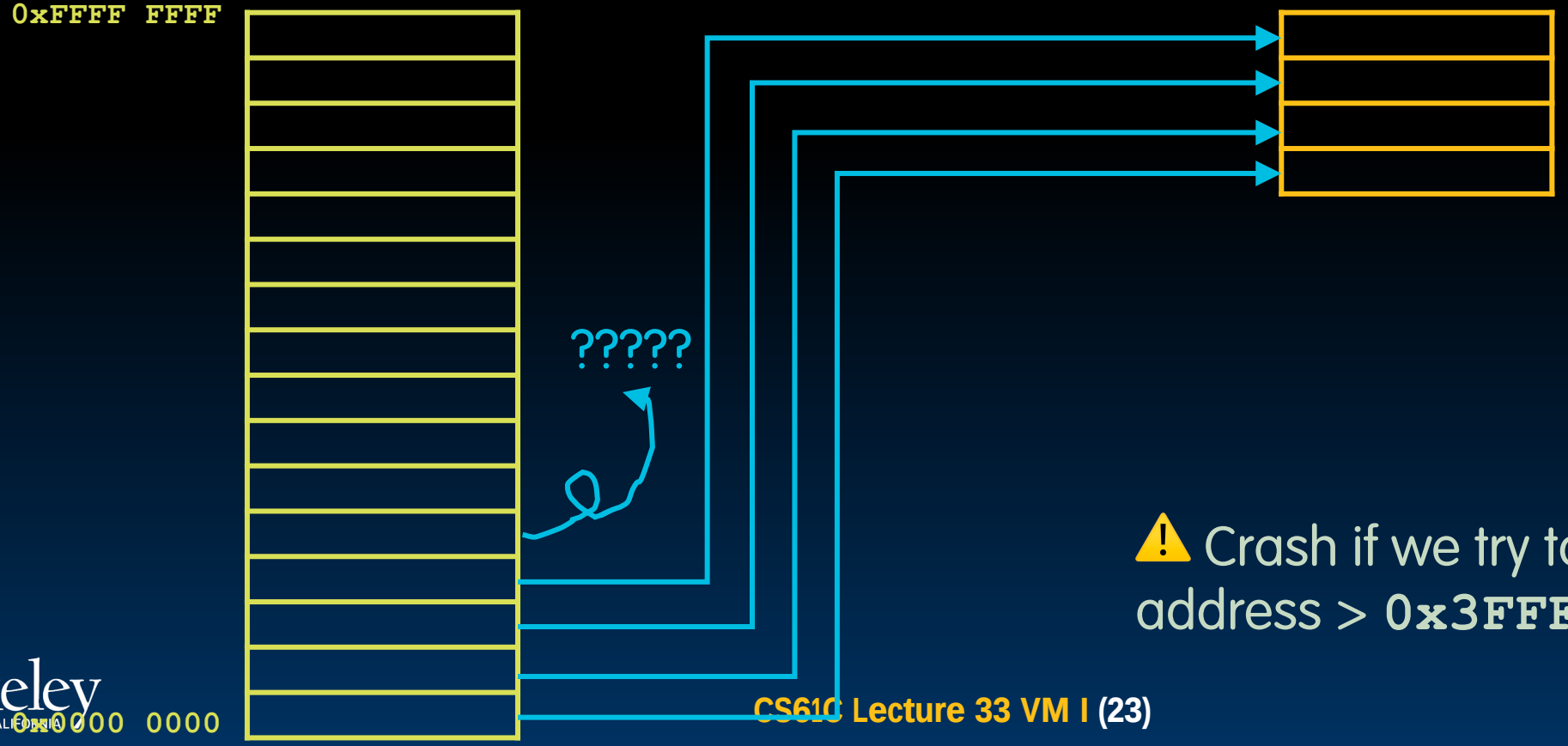
- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

The Case for Virtual Memory (1/2)

1. **What if main memory is smaller than the program address space?**

RV32I provides a 32-bit address space.
→ 2^{32} B = 4 GiB addressable memory

Suppose RAM is 1GiB.
→ 2^{30} B addressable memory.



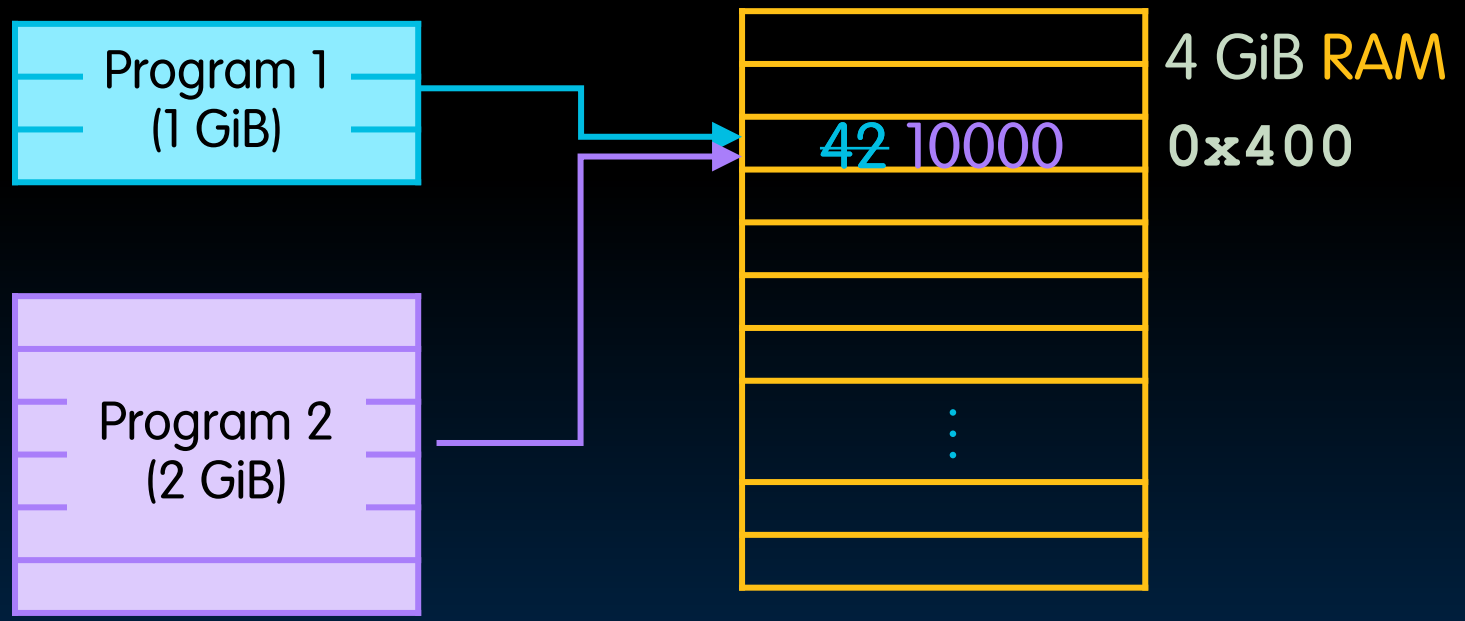
⚠ Crash if we try to access an address > **0x3FFF FFFF**!

The Case for Virtual Memory (2/2)

1. What if main memory is smaller than the program address space?
2. **What if two programs access the same memory address?**

Program 1 stores your bank account balance at address `0x400`

Program 2 stores your video game score at address `0x400`



⚠ If all processes can access any 32-bit memory address, they can corrupt/crash others.

- Need *protection* (i.e., isolation) between processes.

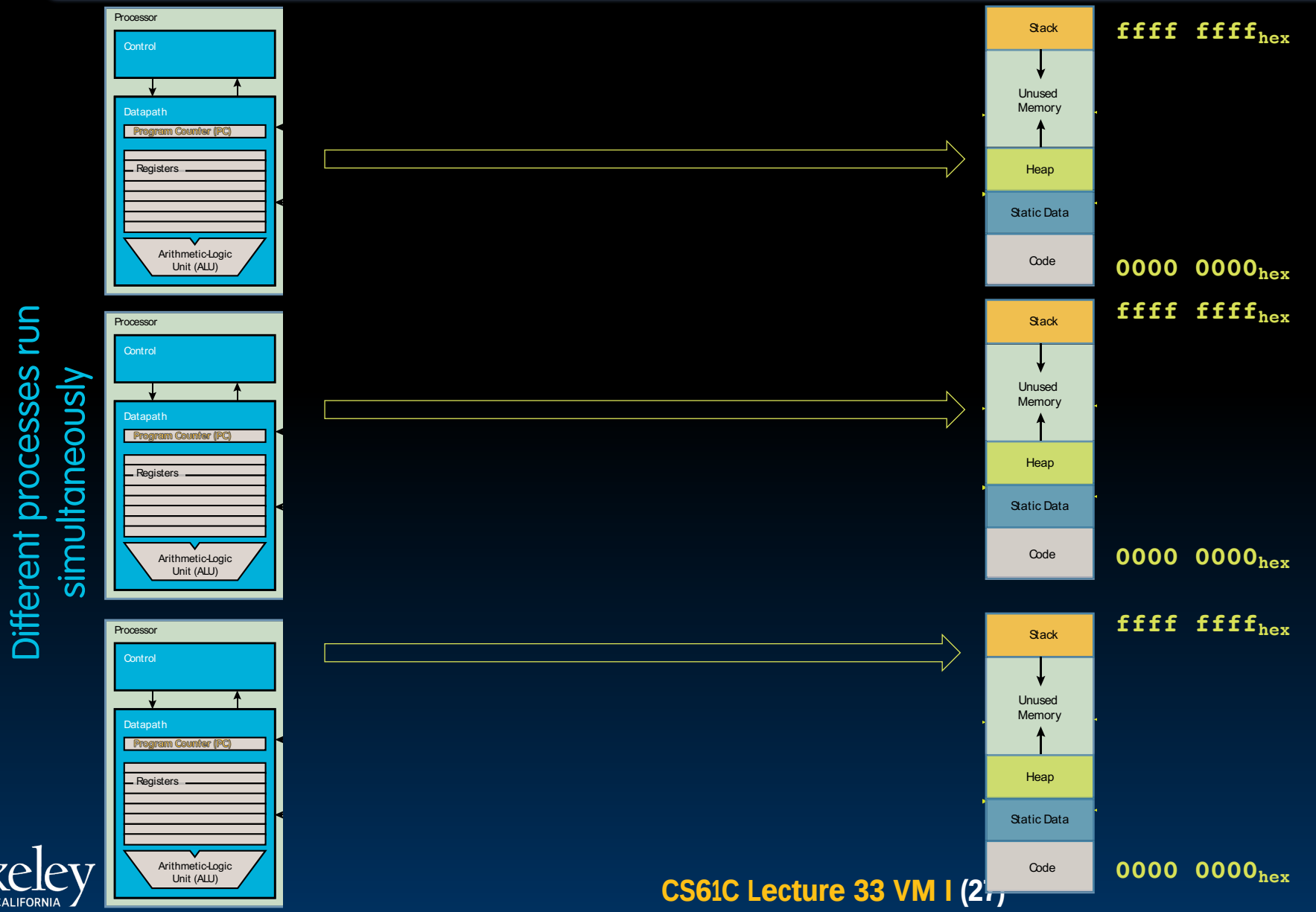
Virtual Memory

- Virtual memory **is the next level in the memory hierarchy:**
  - Give each process the *illusion* of a full memory address space that it has completely for itself.
  - Under the hood: working set of *pages* reside in main memory; other pages are in disk.
- **Benefits:**
  - *Demand paging* provides the ability to run programs larger than the primary memory (DRAM).
  - OS can share memory and *protect* programs from each other.
  - Hides differences between machine configurations.
- **Today, more important for protection than space management.**
  - (Historically, virtual memory predates caches.)

Virtual vs. Physical Addresses

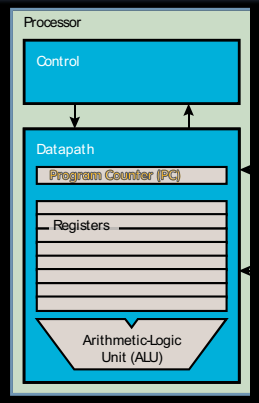
- Address Space: **set of addresses for all available memory locations.**
  - Now, two kinds of memory addresses!
- Virtual Address Space
  - Set of addresses that the user program knows about
- Physical Address Space
  - Set of addresses that map to actual physical locations in memory
  - Hidden from user applications
- **For each program, a memory manager maps (translates) between these two address spaces.**

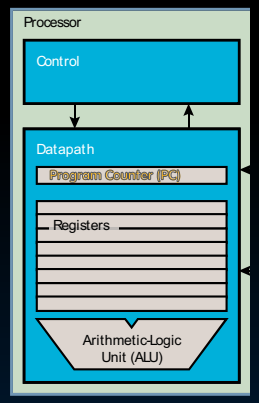
Virtual Address Space Illusion

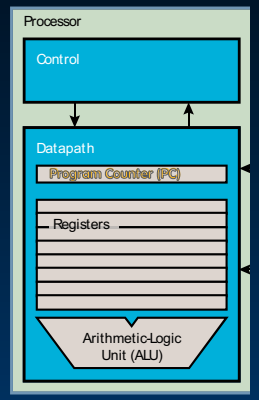


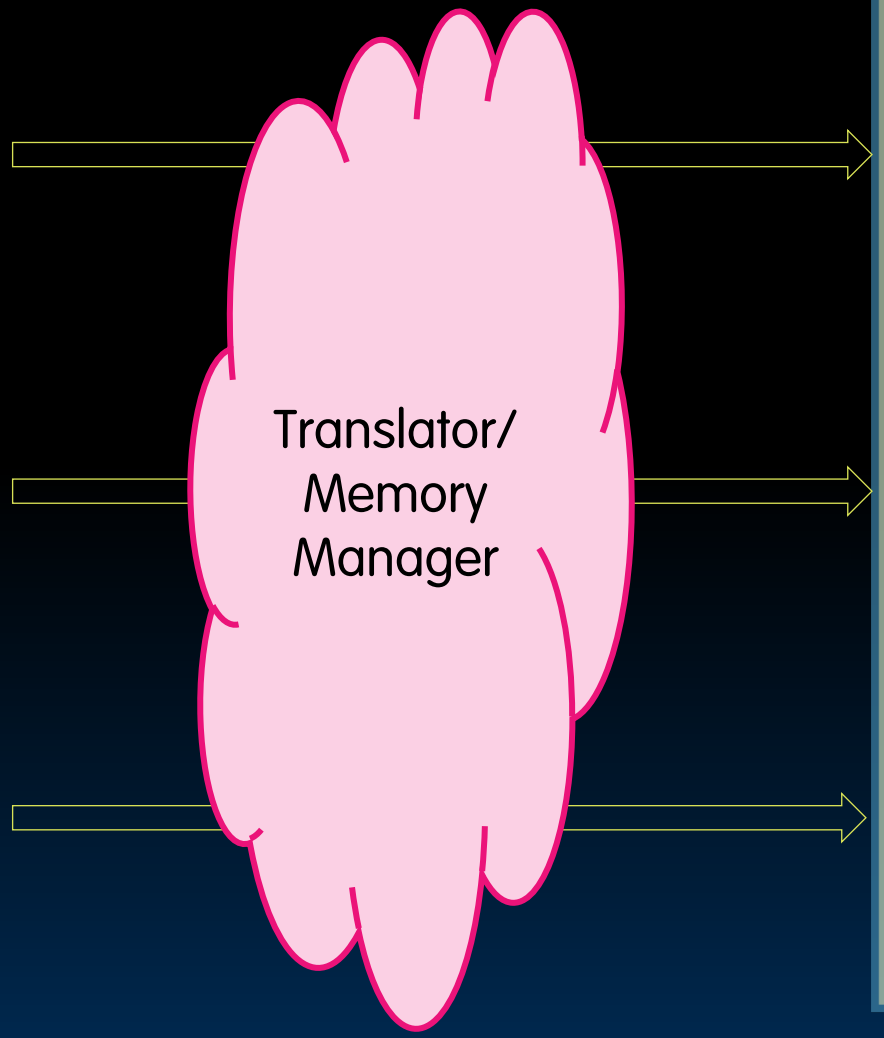
Processes use virtual addresses.
Many processes, all using same (conflicting) addresses

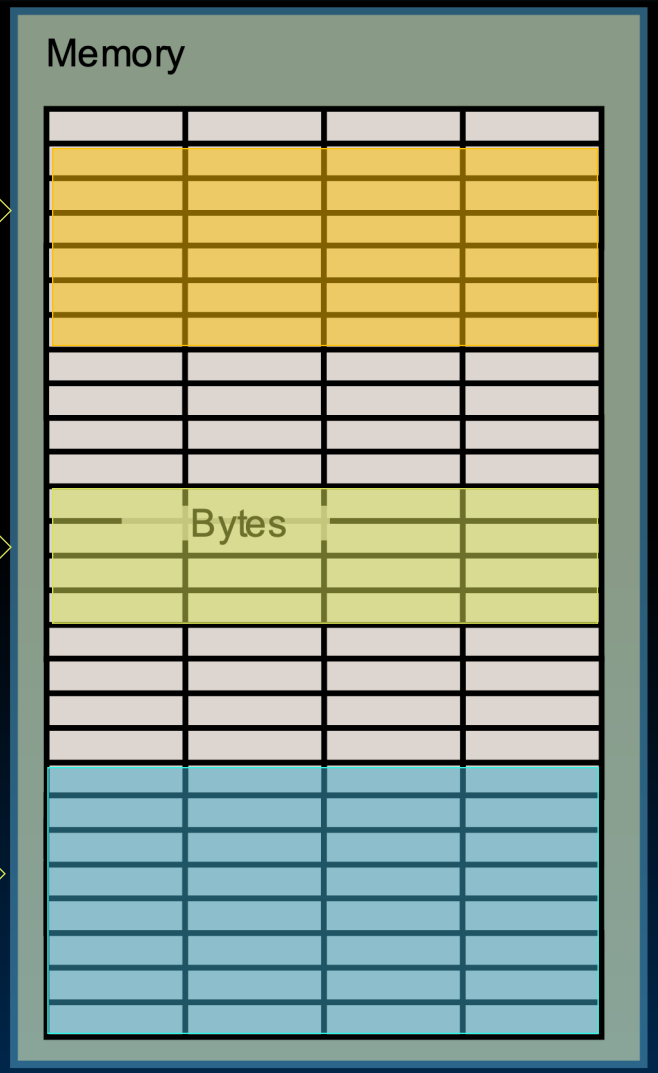
Conceptual Memory Manager in OS









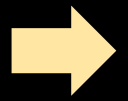


$7fff\ ffff_{hex}$

$0000\ 0000_{hex}$

Memory uses physical addresses.

CS61C Hive Machines

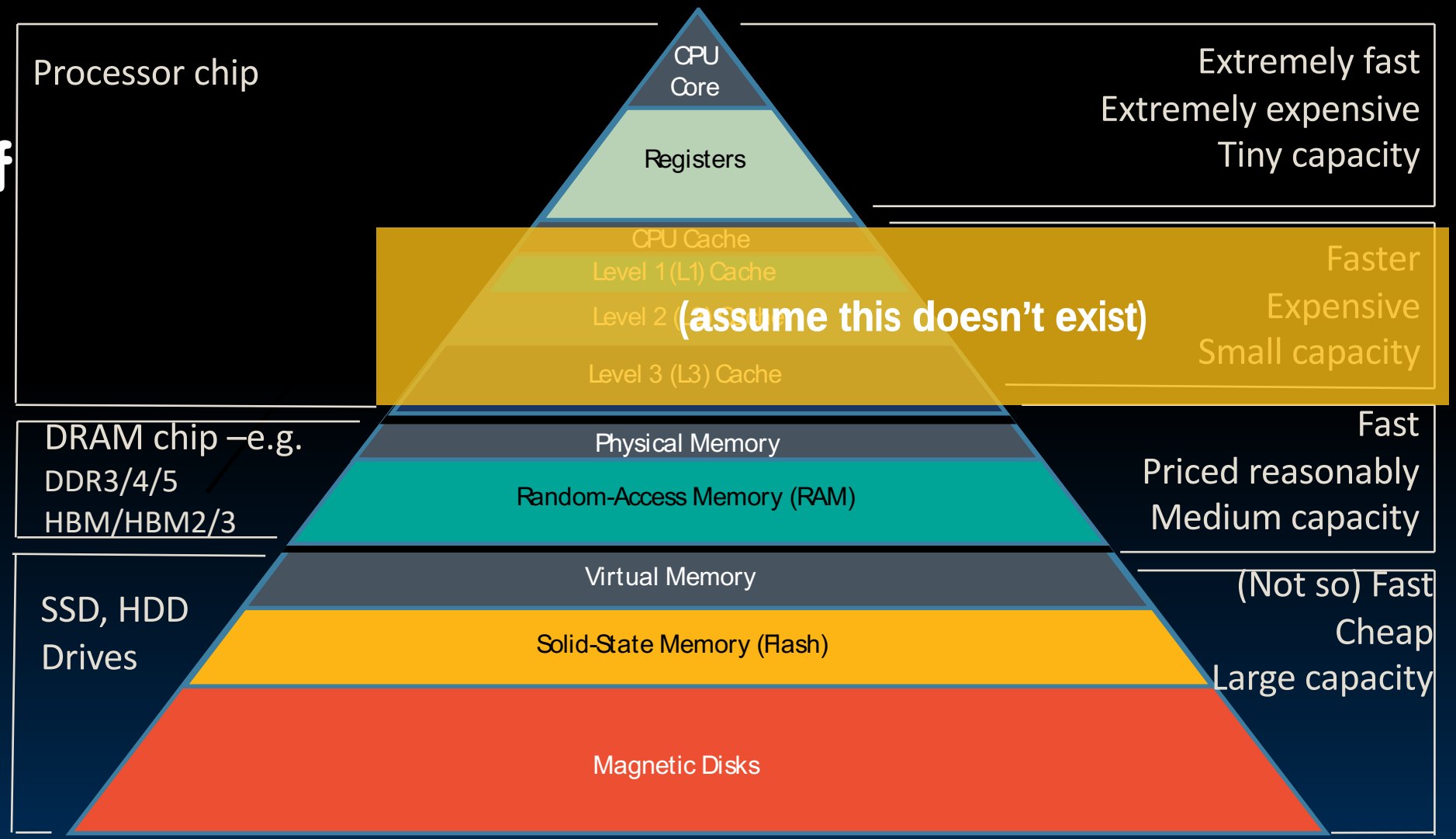


```
(00:18:45 Mon Nov 07 2022 cs61c-tab@hive2 Linux x86_64)
[~ $ cat /proc/cpuinfo
address sizes   : 39 bits physical, 48 bits virtual
```

⚠ Assume Caches Don't Exist For Now ⚠

Virtual Memory **is much easier to understand if we assume** no caches**.**

- We'll reintroduce caches along with *Translation Lookaside Buffers (TLBs)* soon.



Paged Memory

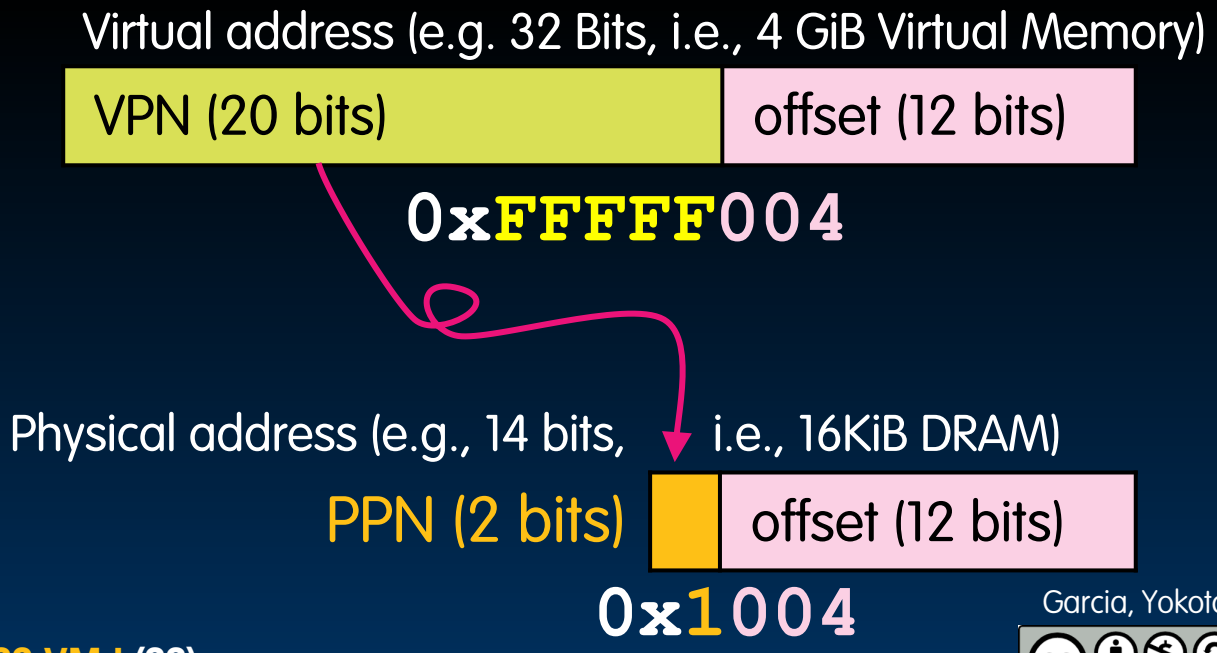
- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

OS Virtual Memory Management Responsibilities

1. **Map virtual addresses to physical addresses.**
2. **Use both memory and disk.**
   - Give illusion of larger memory by storing some content on disk.
   - Disk is usually much larger and slower than DRAM.
3. **Protection:**
   - Isolate memory between processes.

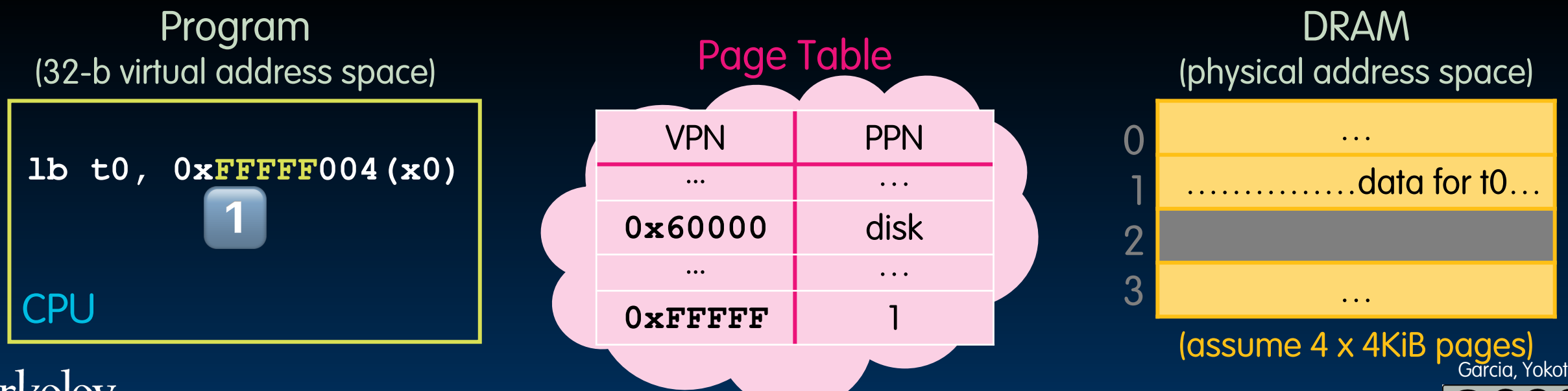
Paged Memory

- **The concept of "paged memory" dominates:**
  - Physical memory (DRAM) is broken into *pages*.
  - A disk access loads an entire page into memory.
  - Typical page size: 4 KiB+ (on modern OSs). Let's assume it's 4KiB…
    - Need 12 bits of *page offset* to address all 4 KiB.
- **Memory translation maps** Virtual Page Number (VPN) **to a** Physical Page Number (PPN).



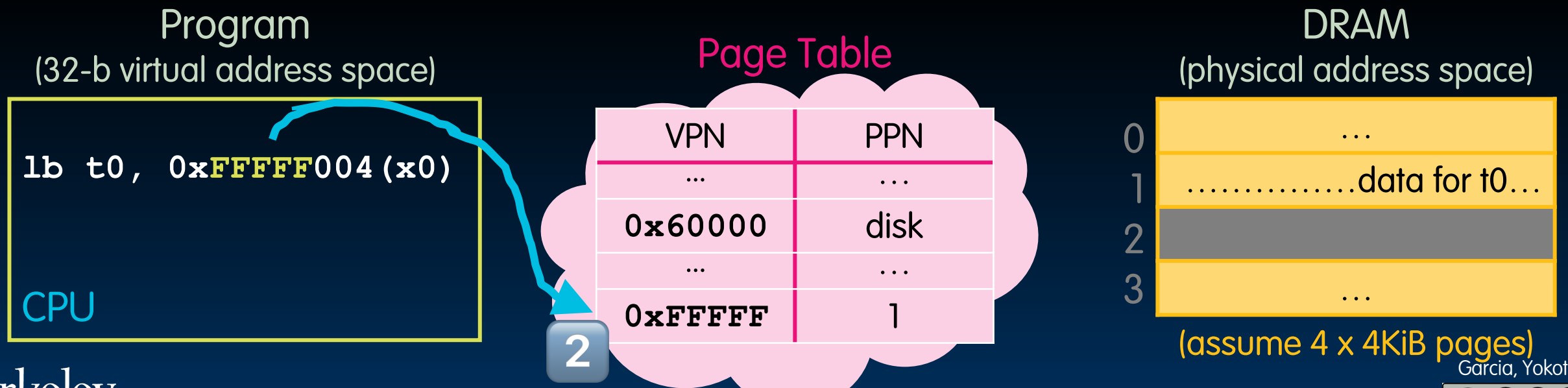
Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**



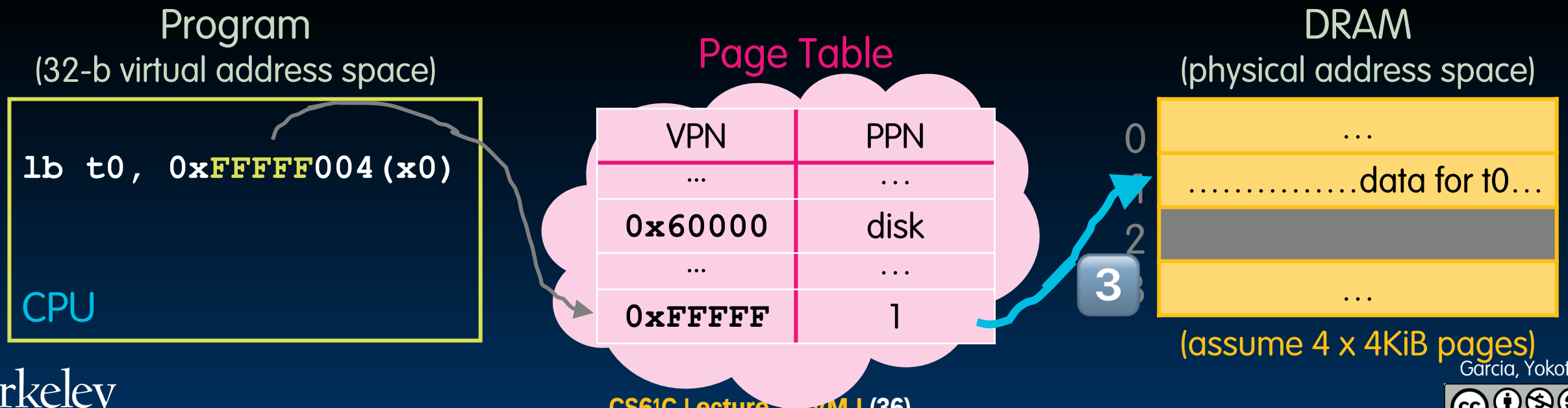
Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA (e.g., top 20 bits if page size 4KiB = 2^{12} B)
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**



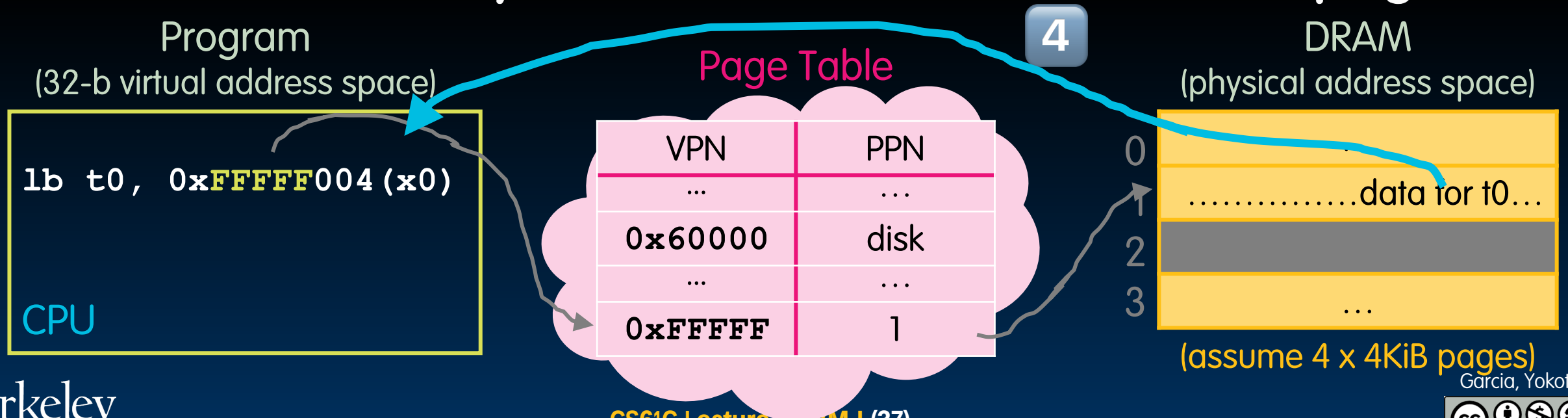
Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA)**.**
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. ~~**If the physical page is not in memory, then OS loads it in from disk.**~~
4. **The OS reads memory at the** PA **and returns the data to the program.**



Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the PA and returns the data to the program.**



Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**

Program
(32-b virtual address space)

```
lb t0, 0xFFFFF004(x0)
lb t1, 0x60000030(x0)
```

1

CPU

Page Table

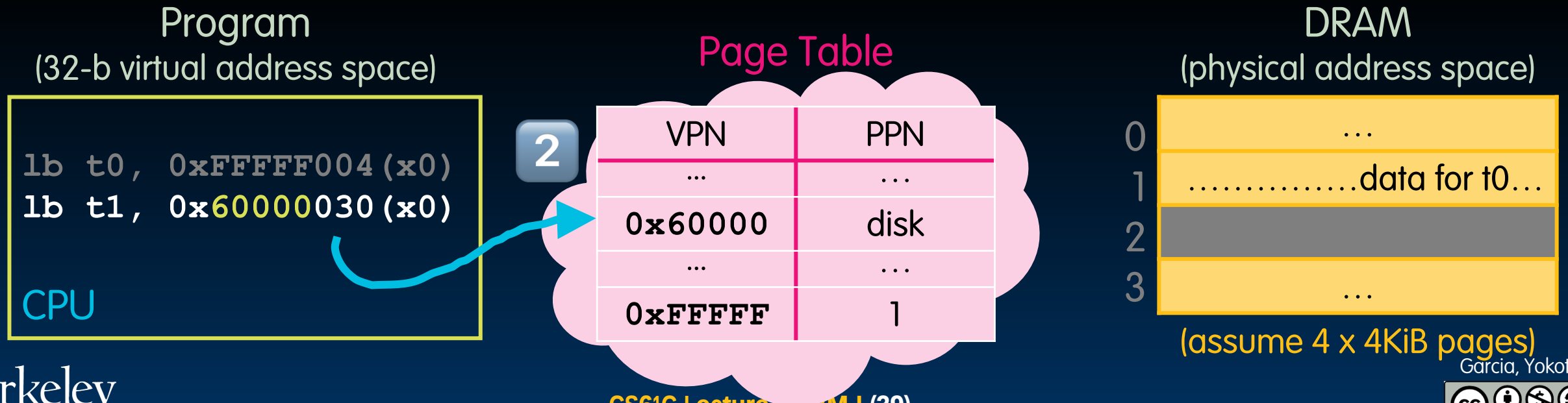
VPN	PPN
…	…
0x60000	disk
…	…
0xFFFFF	1

DRAM
(physical address space)

(assume 4 x 4KiB pages)

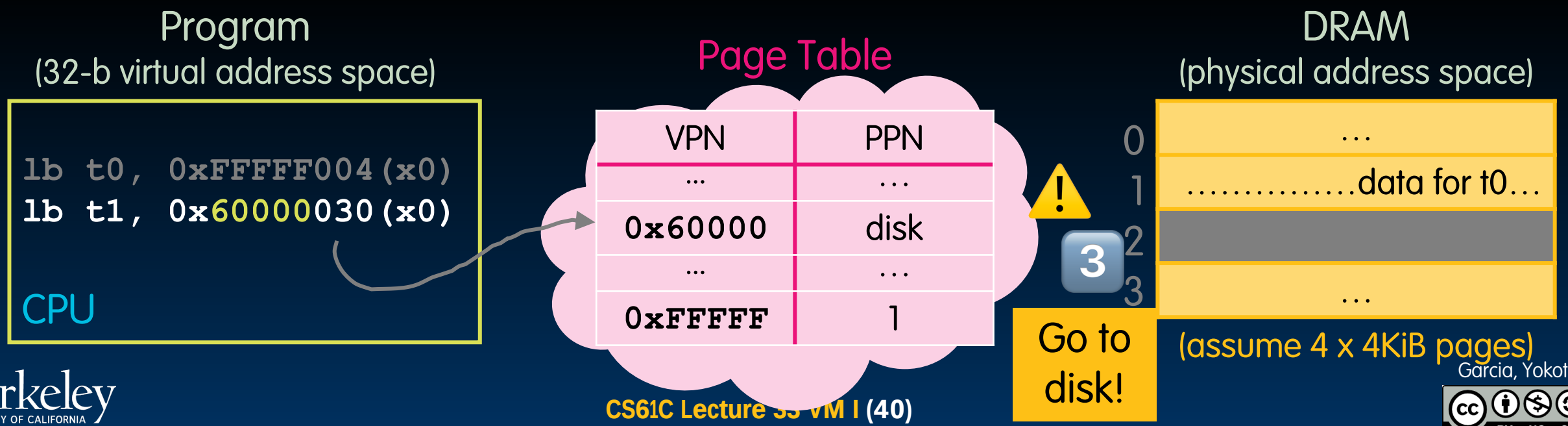
Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA (e.g., top 20 bits if page size 4KiB = 2^{12} B)
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**



Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**

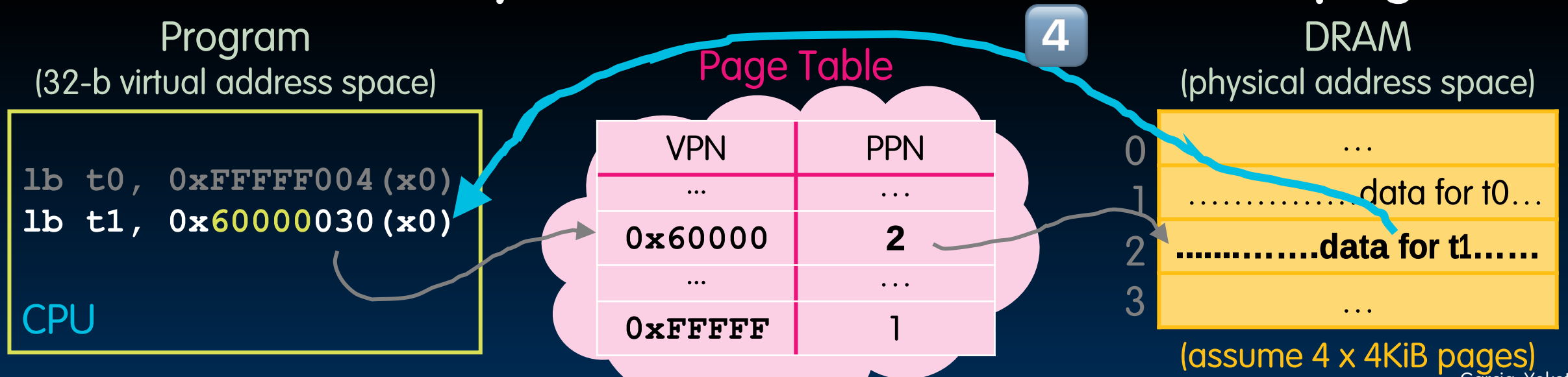


Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the** PA **and returns the data to the program.**

Translation: How a Program Accesses Memory

1. **Program executes a load specifying a** virtual address (VA).
2. **Computer translates** VA **to the** physical address (PA) **in memory.**
   - Extract virtual page number (VPN) from VA, e.g., top 20 bits
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **If the** physical page **is not in memory, then OS loads it in from disk.**
4. **The OS reads memory at the PA and returns the data to the program.**

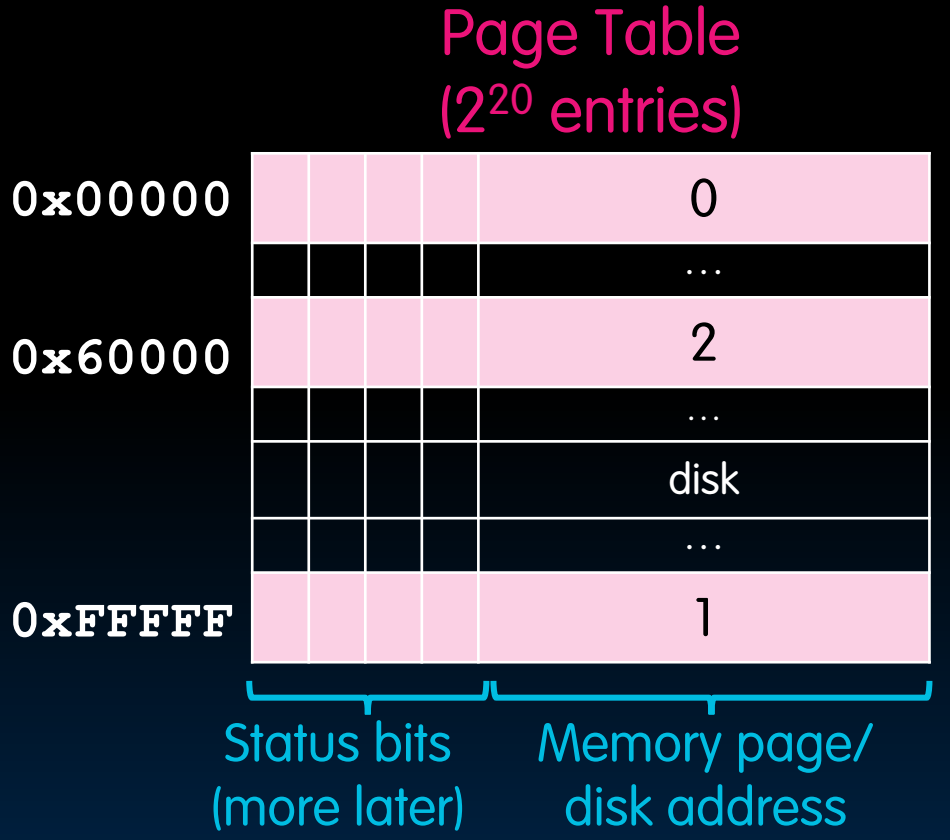


What Do Page Tables Look Like?

- **E.g., 32-bit virtual address space, 4-KiB pages**
  - 2^{32} virtual addresses / (2^{12} B/page)
    = 2^{20} virtual page numbers (1 Mebi pages)
- **One Page Table per process:**
  - One entry per virtual page number.
  - Entry has physical page number (or disk address) as well as status bits.

Note: A Page Table is NOT a cache!!

- A Page Table does not have data! It is a lookup table.
- All VPNs have a valid entry.
  - But if it helps you, "no tags; index is VPN"



Page Table Details I

- The Computer
- OS Basics: Context Switching
- Physical Memory and Disk Storage
- Virtual Memory and Virtual Addresses
- Paged Memory
- Page Table Details I

OS Virtual Memory Management Responsibilities

✅ 1. **Map virtual addresses to physical addresses.**

✅ 2. **Use both memory and disk.**

- Give illusion of larger memory by storing some content on disk.
- Disk is usually much larger and slower than DRAM.

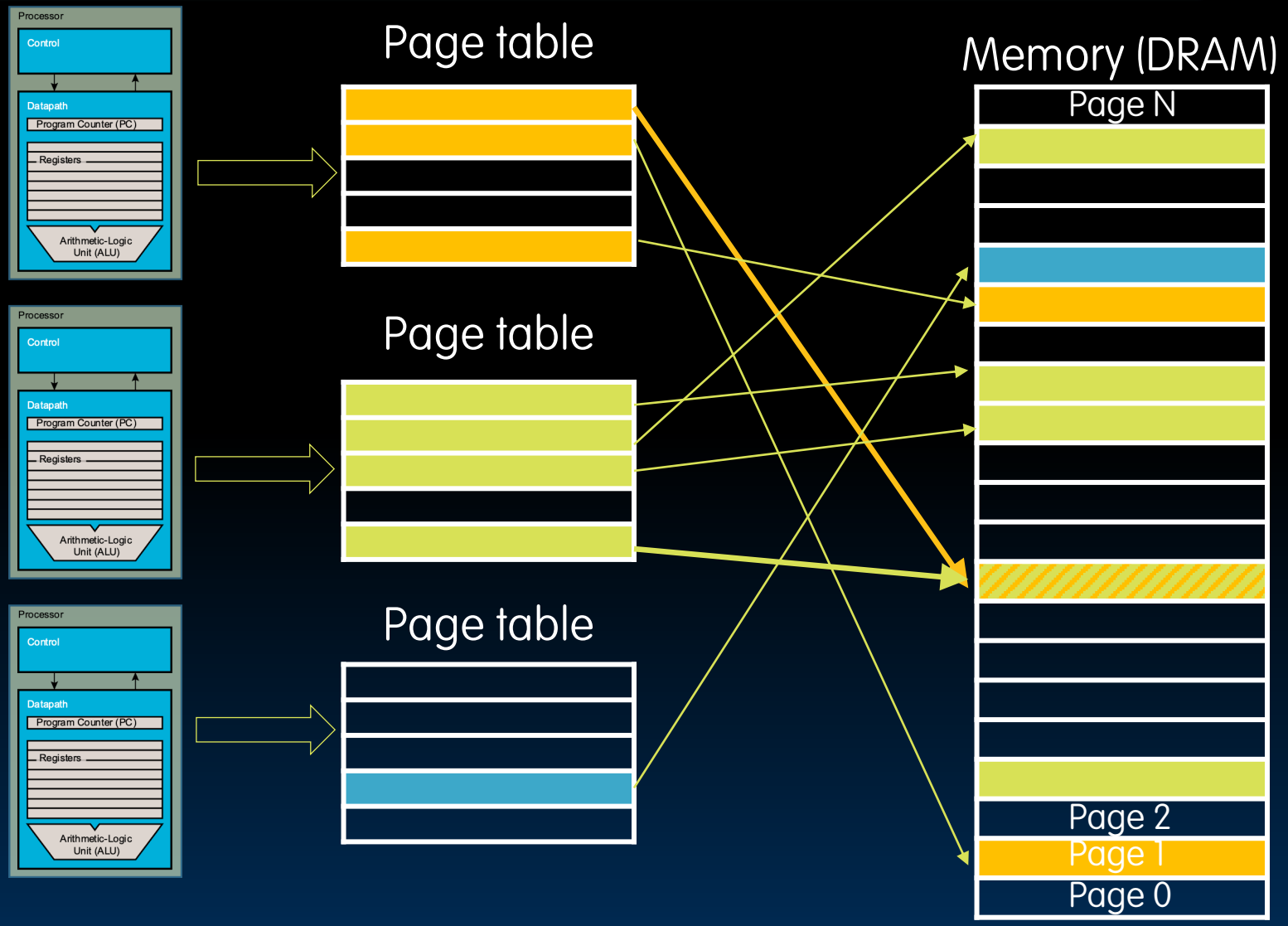
❓ 3. **Protection:**

- Isolate memory between processes.
- Each process gets dedicated "private" memory.
- Errors in one program won't corrupt memory of other programs.
- Prevent user programs from messing with OS's memory.

What if process tries to modify instructions or system data?

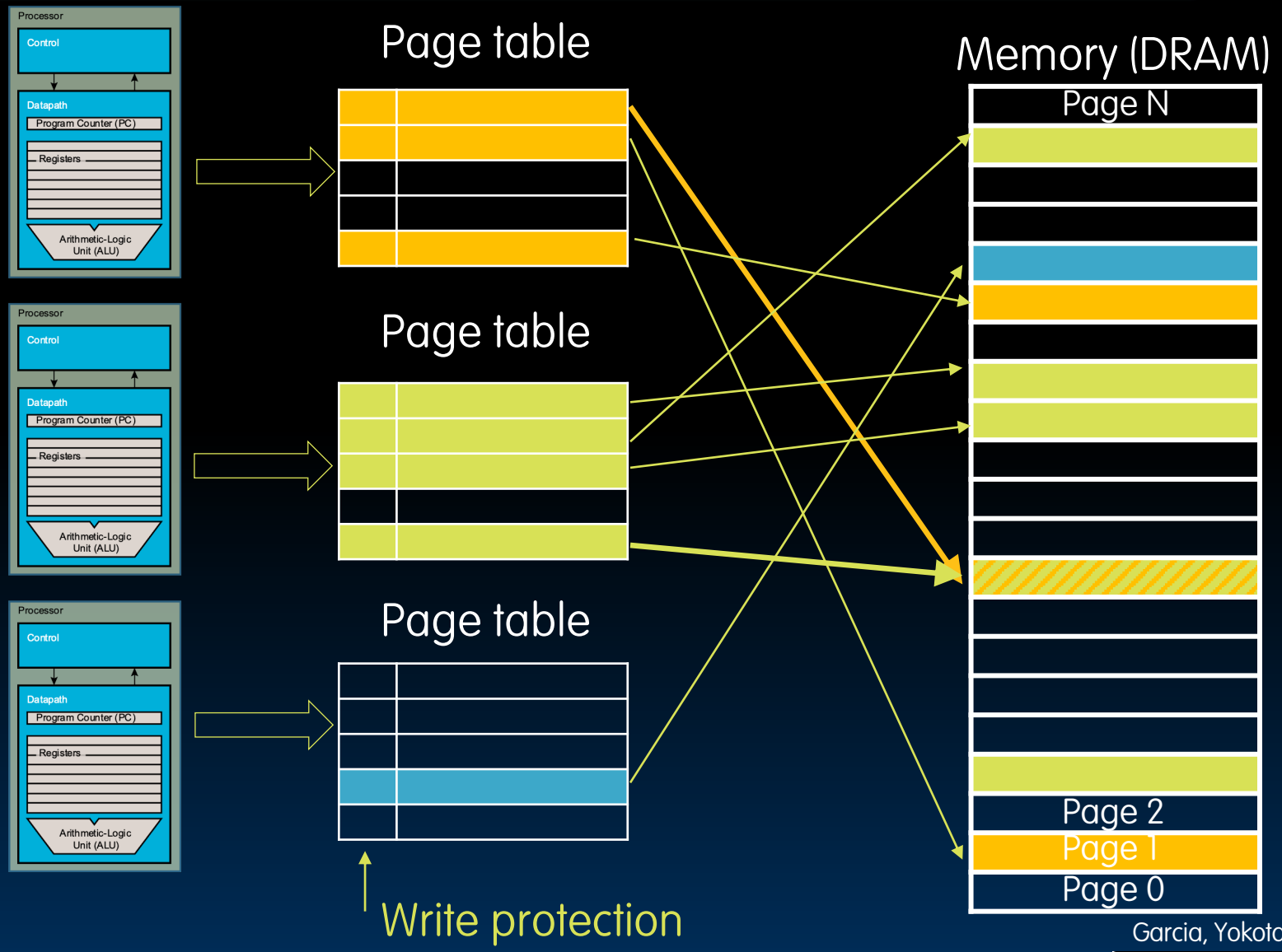
Protection with Page Tables (1/2)

- **Each process has a dedicated page table.**
  - OS keeps track of which process is active.
- Isolation: **Assign processes different pages in DRAM**
  - Prevents accessing other processors' memory
  - Page tables managed by OS
- **Sharing is also possible:**
  - OS may assign same physical page to several processes, e.g., system data



Protection with Page Tables (2/2)

- **Page Table Entry also includes a write protection bit.**
- **If on, then page is "protected":**
  - e.g., program code, system data, etc.
  - Writing to a protected page triggers an exception.
  - Exceptions are handled by OS. (more later)



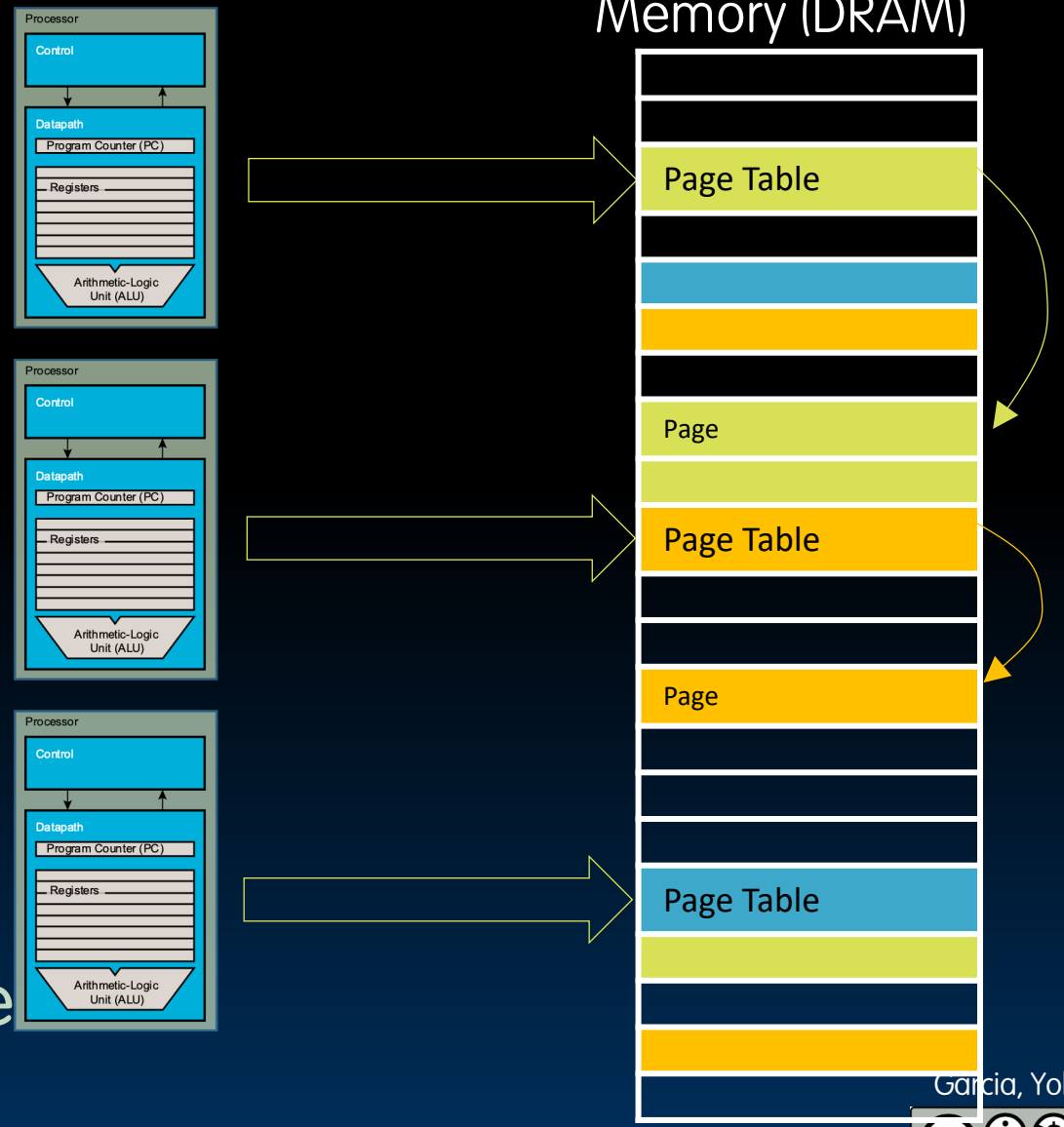
Page Tables Are Stored in Memory (1/2) (for next time)

- **E.g., 32-Bit virtual address space, 4-KiB pages**
  - Single page table size (suppose each entry is 4B, including status bits):
    - 4×2^{20} Bytes = 4-MiB
    - 0.1% of 4-GiB memory. Not bad. But much too large for a cache!
- **For now, store page tables in memory (DRAM).**
  - Caveat: *Two* (slow) memory accesses per **`lw/sw`** on cache miss!

Page Tables Are Stored in Memory (2/2)

(for next time)

- **Caveat: `lw/sw` then requires two memory accesses:**
  - Read page table (stored in main memory) to translate to physical address
  - Read physical page, also in main memory
- **To minimize the performance penalty:**
  - Transfer blocks (not words) between DRAM and processor cache
  - Use a cache for frequently used page table entries … (more later, TLB)



And in Conclusion…

- **The OS manages resources across multiple processes, all sharing the same CPU, memory, I/O devices, etc.**
- **Each process operates in virtual memory.**
  - For each process, the OS manages virtual↔physical address translation via page tables.
- **Open questions:**
  - How does the OS "context switch"?
  - Write-back or write-through?
  - How to incorporate caches with virtual memory?