UC Berkeley
Teaching Professor
Lisa Yan

CS61C

Great Ideas
in
Computer Architecture
(a.k.a. Machine Structures)

UC Berkeley
Lecturer
Justin Yokota

Virtual Memory I

cs61c.org

Agenda

Virtual Memory and Virtual Addresses

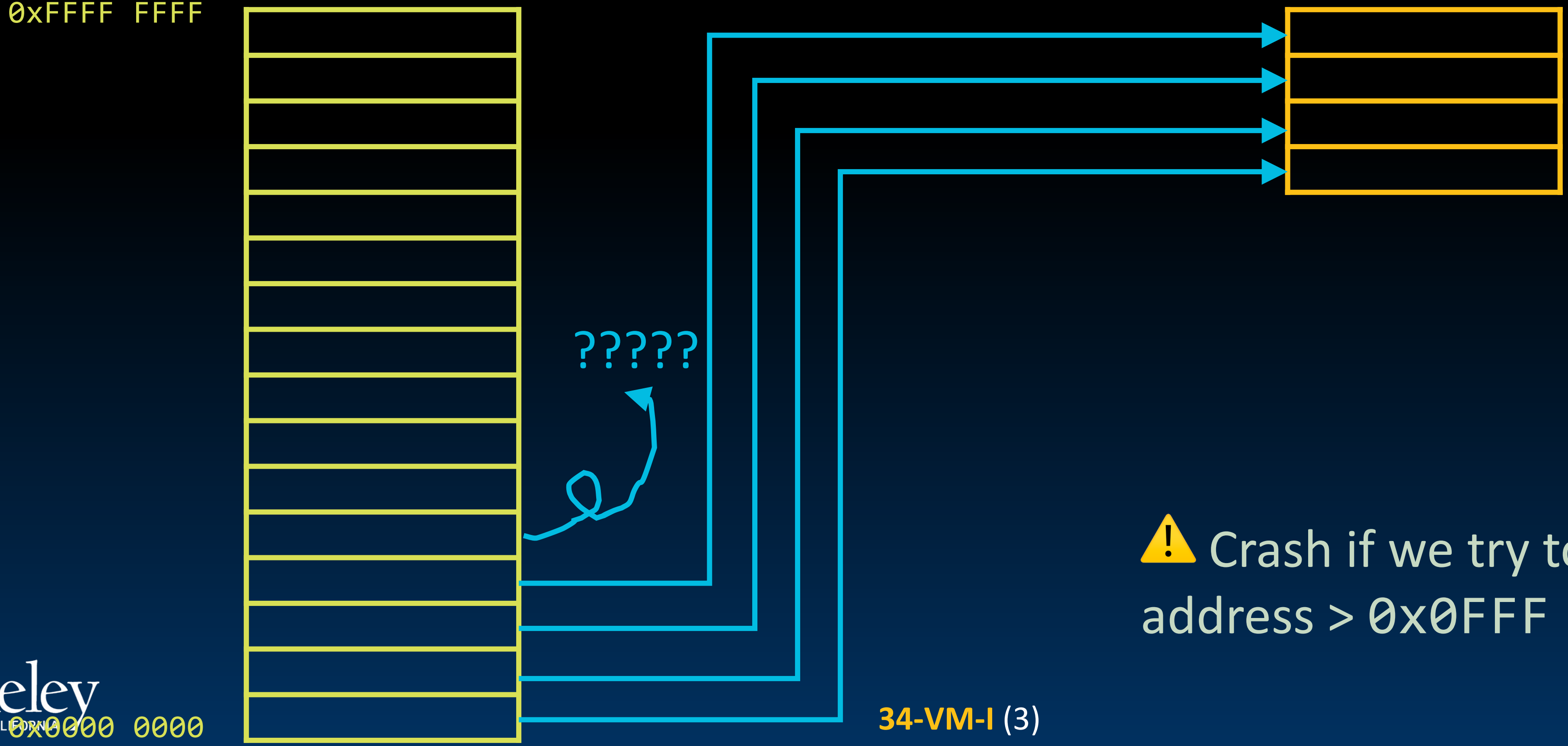
- Virtual Memory and Virtual Addresses
- Paged Memory
- Address Translation
- Practice
- Page Table Details I

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.



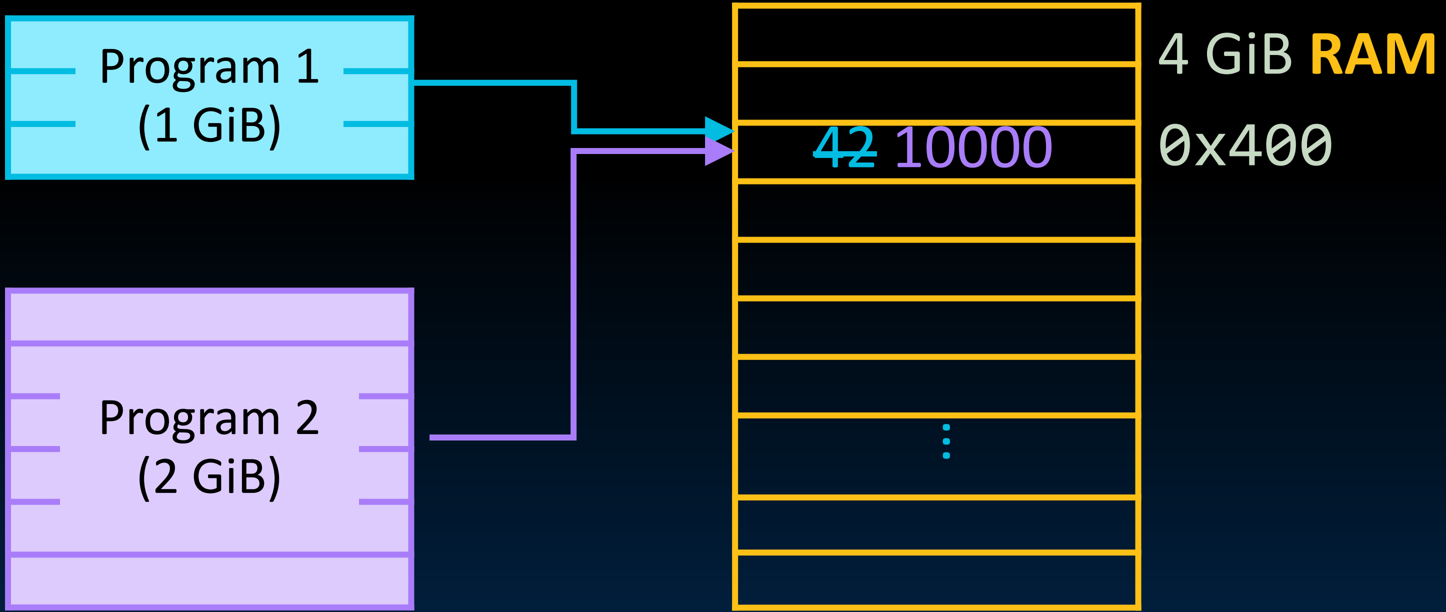
⚠ Crash if we try to access an address > `0x0FFF FFFF`!

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.

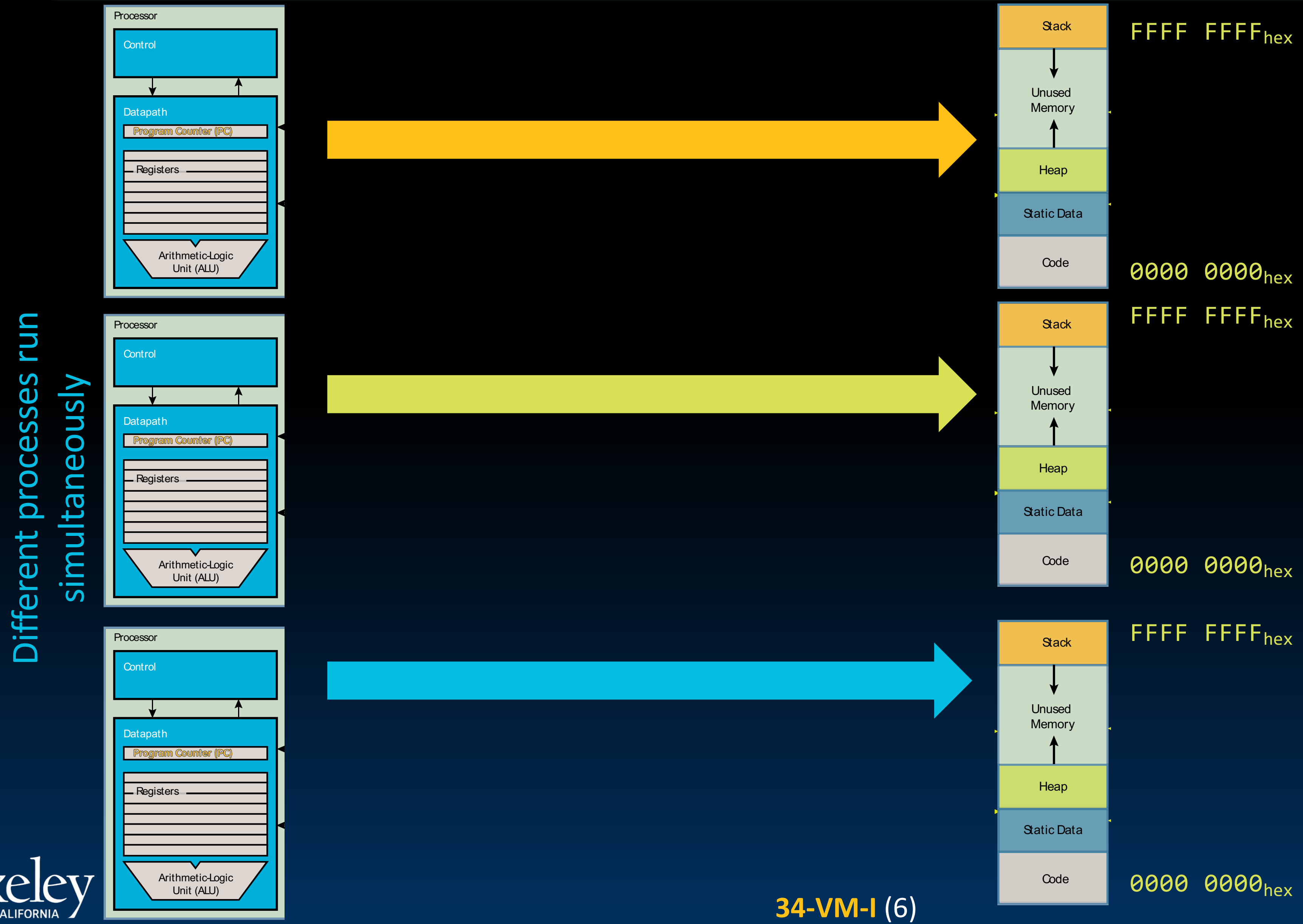
- Need **protection** (i.e., isolation) between processes.

Yan, Yokota

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 on **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.)

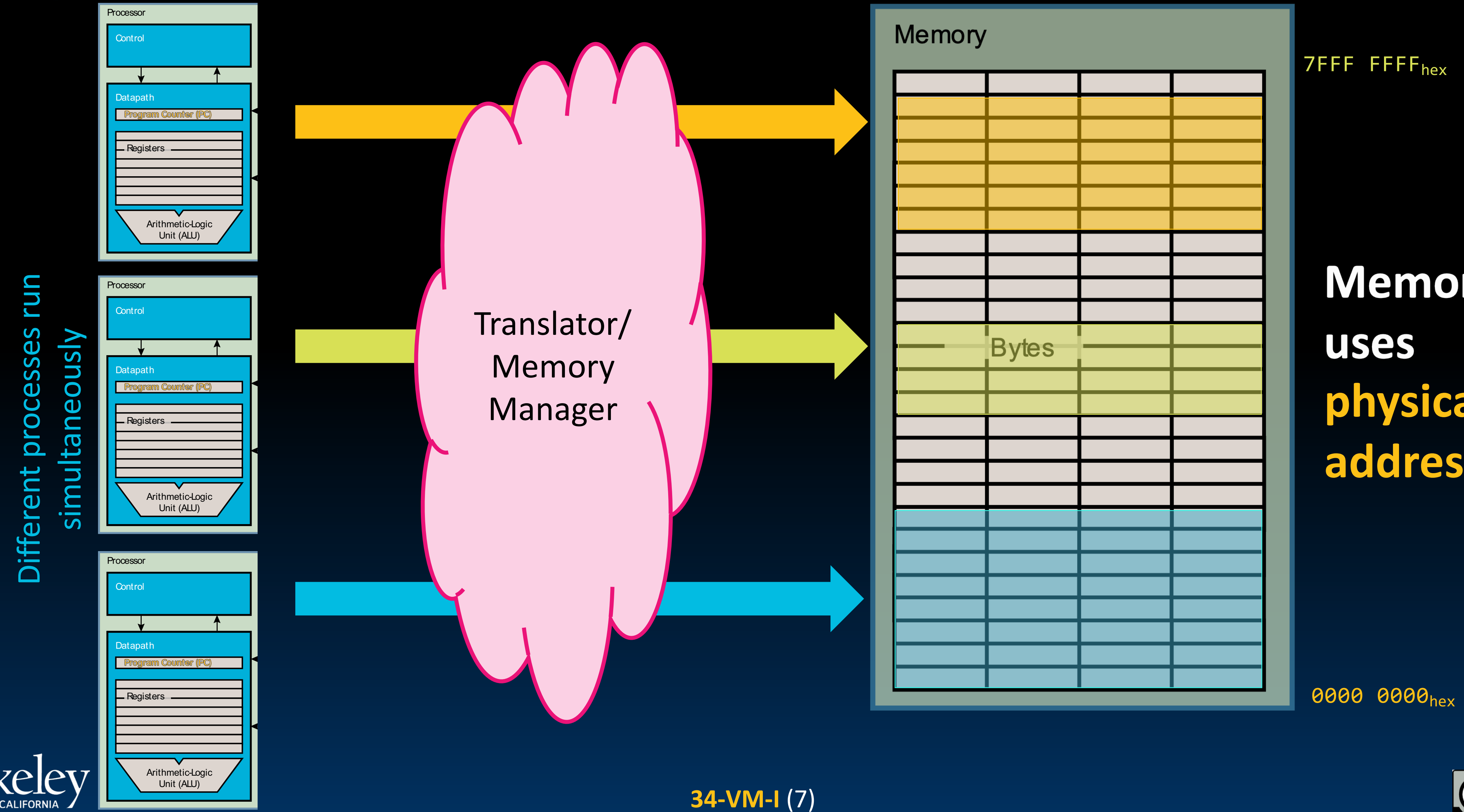
Virtual Address Space Illusion



Processes use virtual addresses.

Many processes, all using same (conflicting) addresses

Conceptual Memory Manager in OS



Memory uses **physical addresses.**

Virtual vs. Physical Addresses

- **Address Space**: set of addresses for all available memory locations.
  - Now, two kinds of memory addresses!

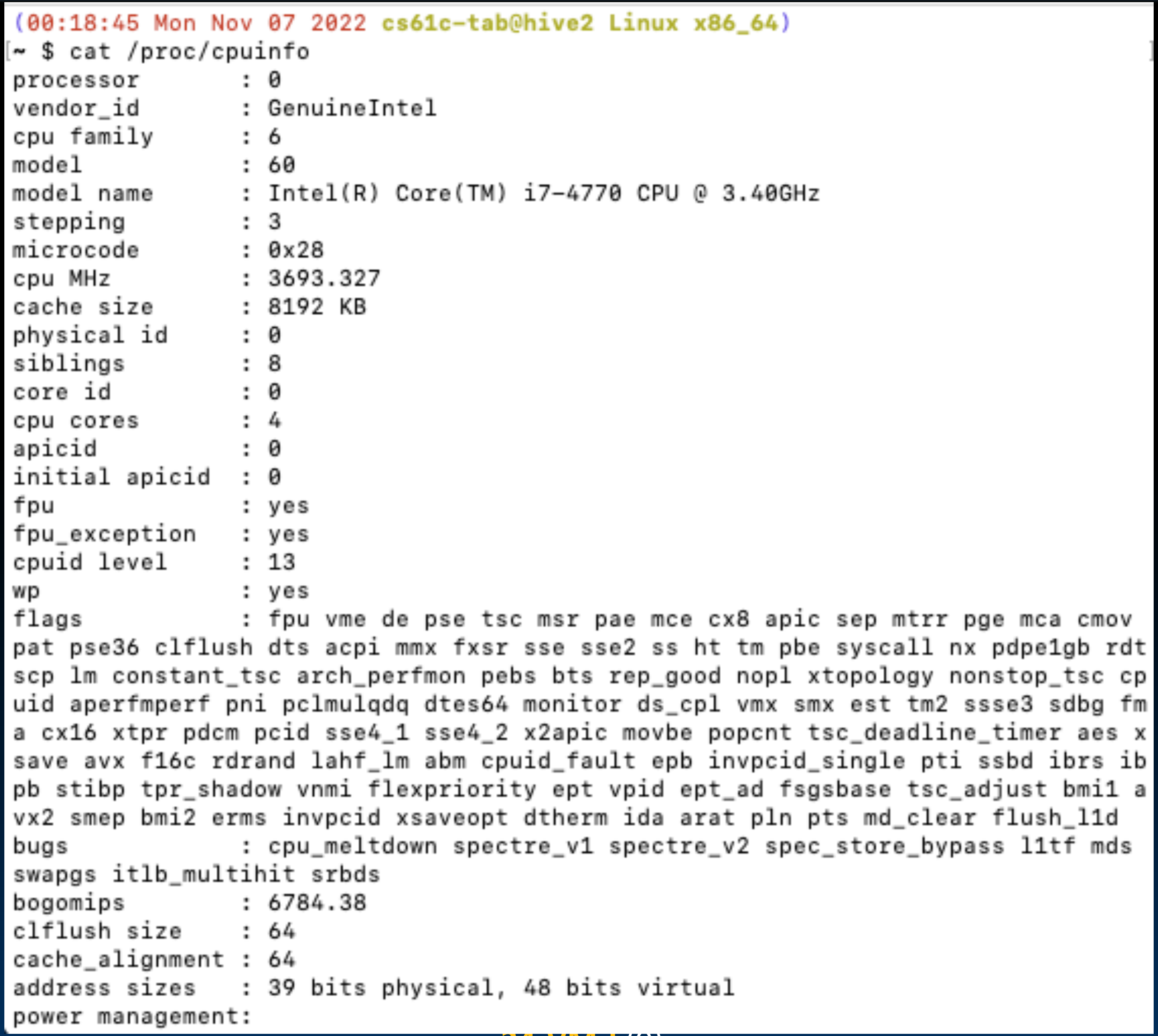
- **Virtual Address Space**
  - Set of addresses that the user program knows about

- **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.**

Yan, Yokota

CS61C Hive Machines

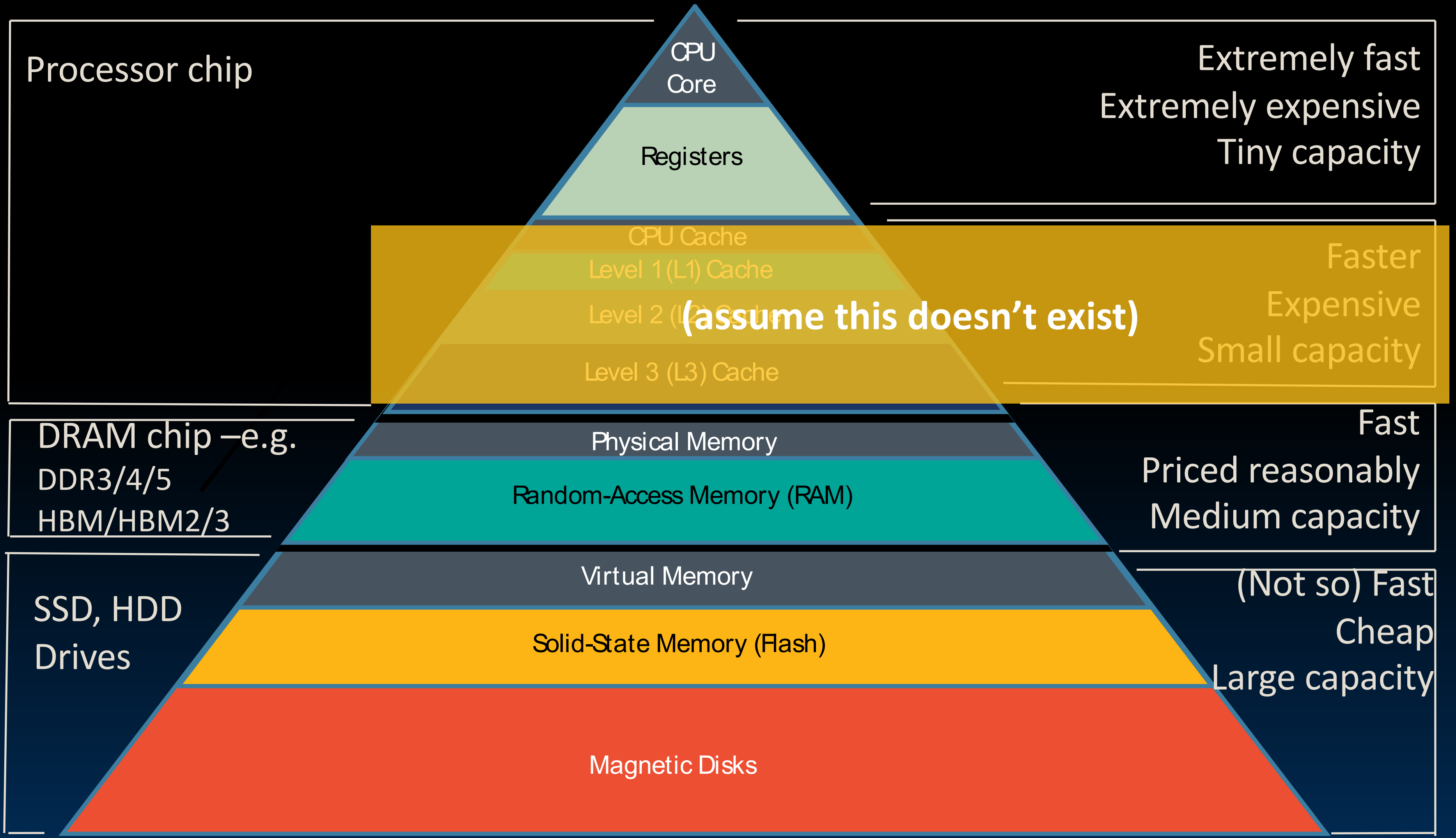


```
(00:18:45 Mon Nov 07 2022 cs61c-tab@hive2 Linux x86_64)
~ $ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping        : 3
microcode       : 0x28
cpu MHz         : 3693.327
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cp
uid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fm
a cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes x
save avx f16c rdrand lahf_lm abm cpuid_fault epb invpcid_single pti ssbd ibrs ib
pb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 a
vx2 smep bmi2 erms invpcid xsaveopt dtherm ida arat pln pts md_clear flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds
bogomips        : 6784.38
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

⚠️ 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.



Yan, Yokota

Agenda

Paged Memory

- Virtual Memory and Virtual Addresses
- Paged Memory
- Address Translation
- Practice
- Page Table Details I

How is the Hierarchy Managed?

[review]

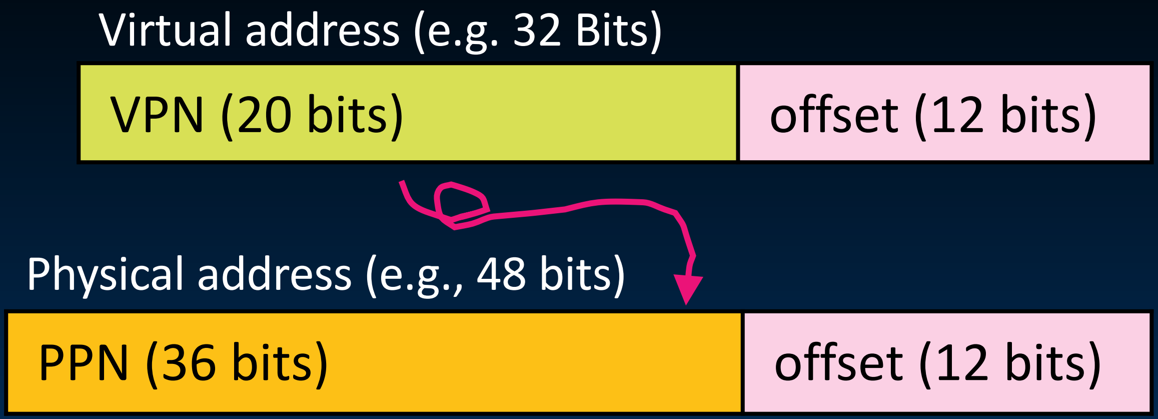
- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware
- main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware (‘translation lookaside buffer’ or TLB)
  - By the programmer (files)

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)
    - Need 12 bits of ***page offset*** to address all 4 KiB bytes.
- If virtual and physical pages are the same size, then memory translation maps **Virtual Page Number (VPN)** to a **Physical Page Number (PPN)**.

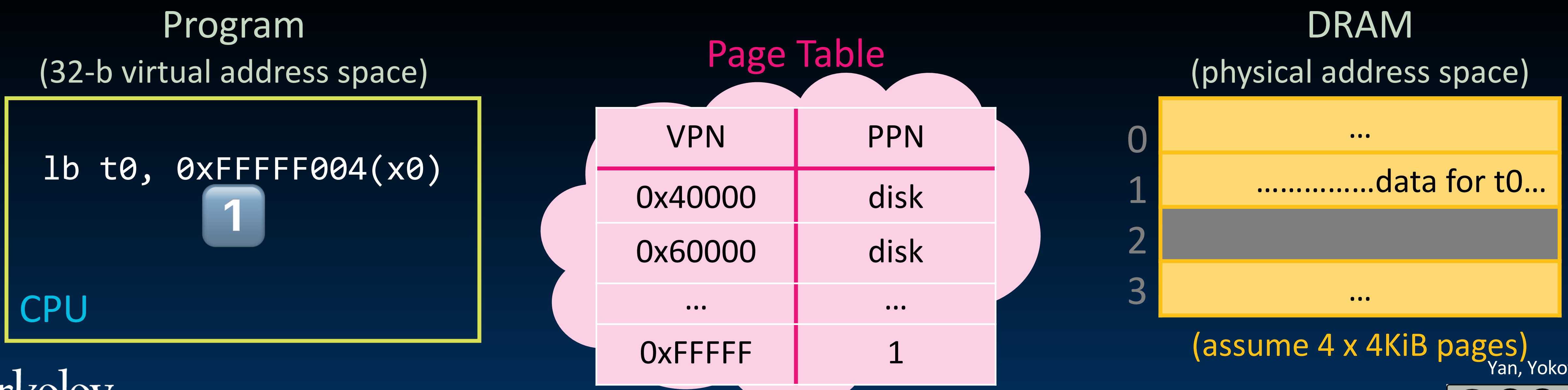

Virtual address (e.g. 32 Bits)
VPN (20 bits)
offset (12 bits)
Physical address (e.g., 48 bits)
PPN (36 bits)
offset (12 bits)

Agenda

Address Translation

Translation: How a Program Accesses Memory

1. Program executes a load specifying a **virtual address (VA)**.

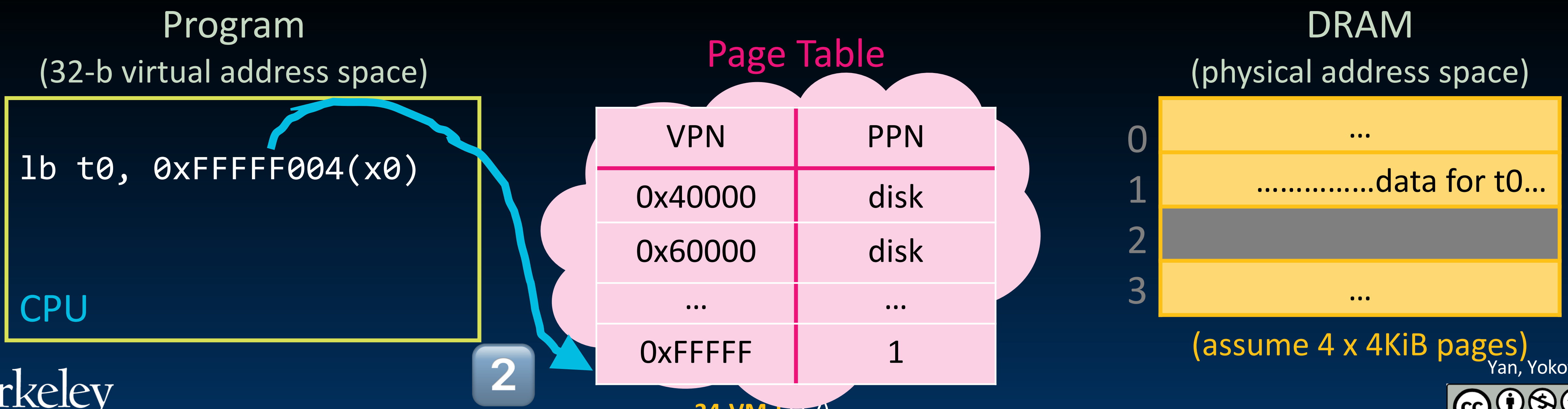


VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

Yan, Yokota

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)



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

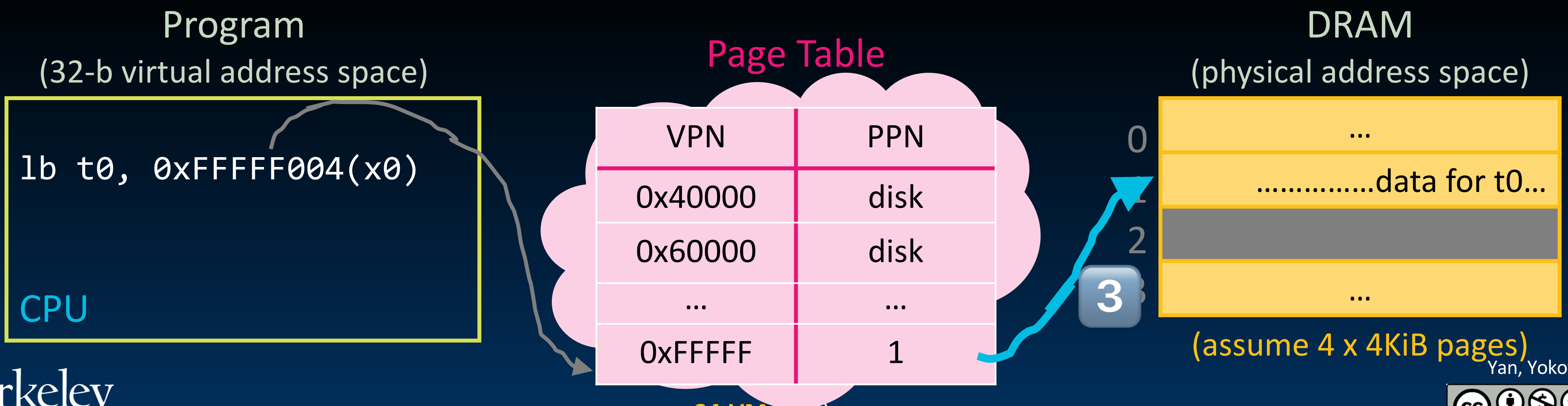
VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

Yan, Yokota

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. **Page Fault**: If **physical page** is not in memory, ~~then OS loads it in from disk.~~

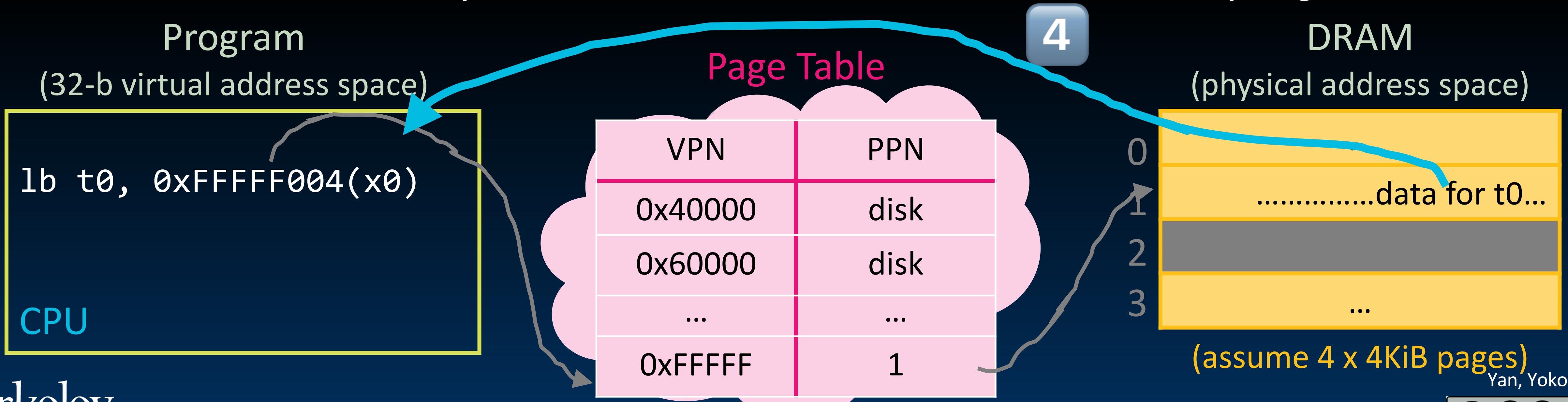
Found in memory!
No page fault



Yan, Yokota

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. **Page Fault**: If **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.

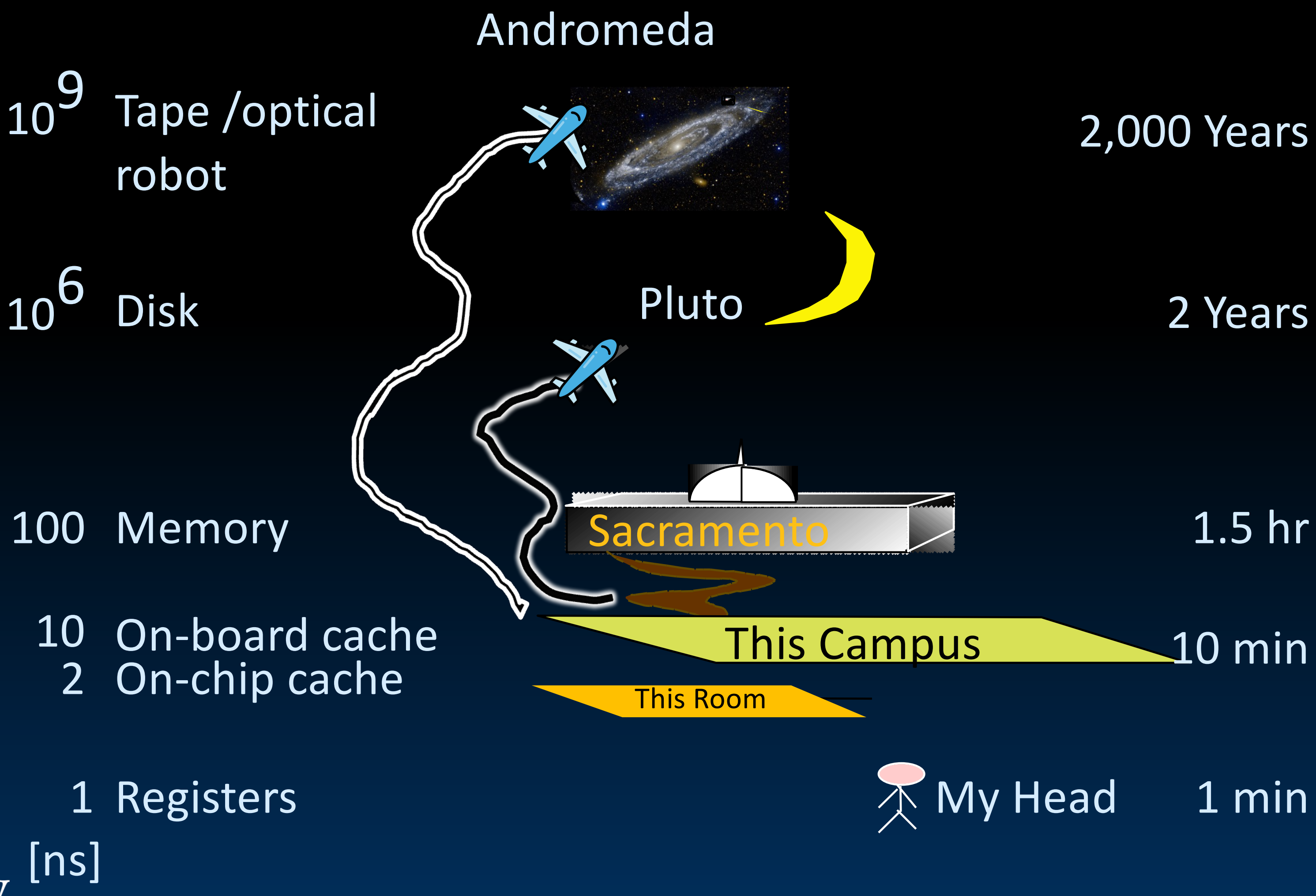


VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

Yan, Yokota

Great Idea #3: Principle of Locality / Memory Hierarchy

Storage Latency Analogy: How Far Away is the Data?



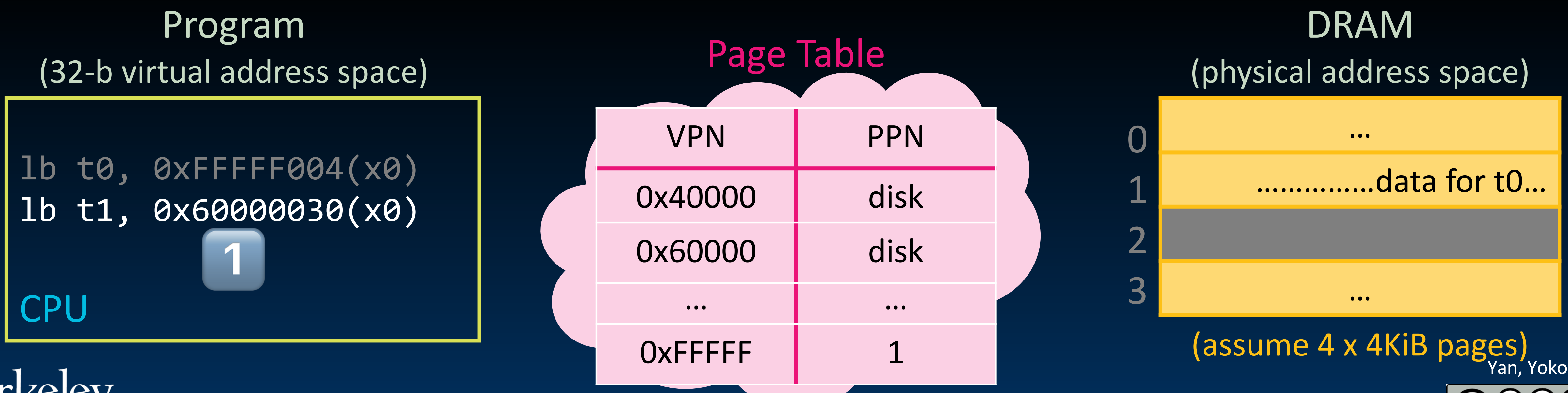
Avoid page faults (disk access).

- **Spatial locality**: Pages are big (≥4KiB)
- **Temporal locality**: Page Table translation

Yan, Yokota

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. **Page Fault**: If **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.



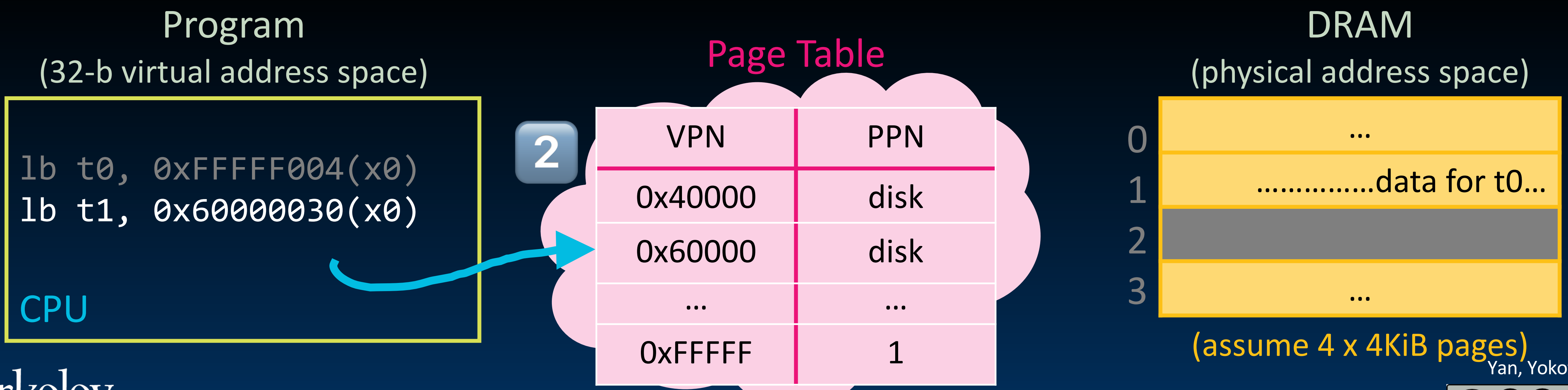
```
lb t0, 0xFFFFF004(x0)
lb t1, 0x60000030(x0)
```

VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

Yan, Yokota

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. **Page Fault**: If **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.

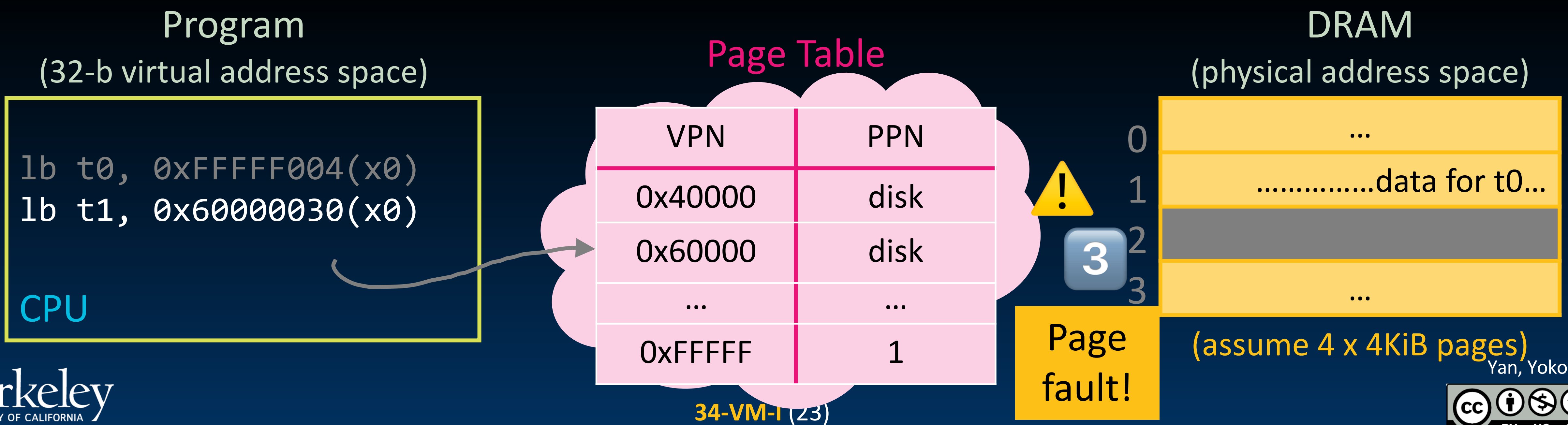


```
lb t0, 0xFFFFF004(x0)
lb t1, 0x60000030(x0)
```

VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

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. **Page Fault**: If **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.



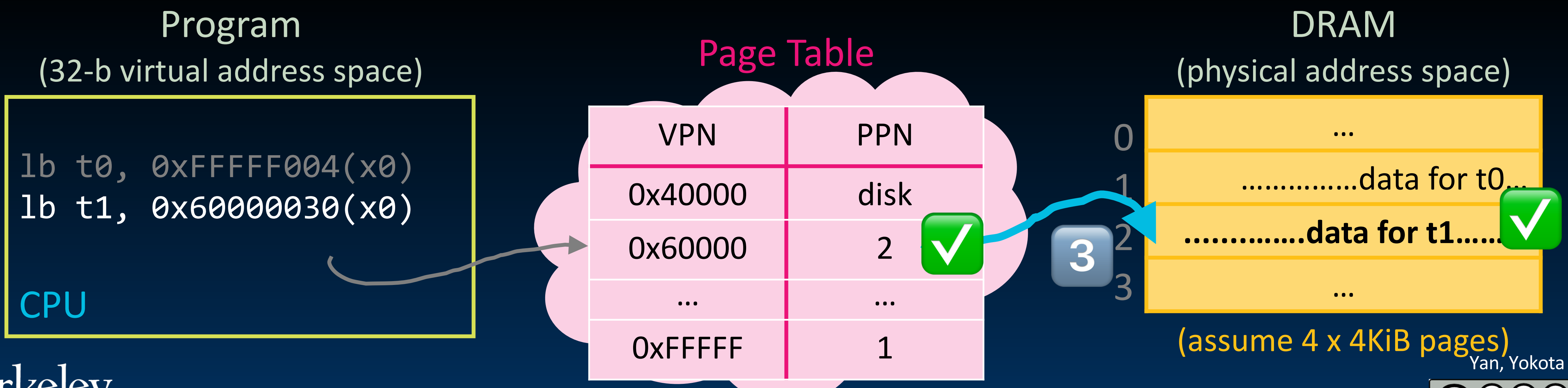
```
lb t0, 0xFFFFF004(x0)
lb t1, 0x60000030(x0)
```

VPN	PPN
0x40000	disk
0x60000	disk
…	…
0xFFFFF	1

Yan, Yokota

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. **Page Fault**: If **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.



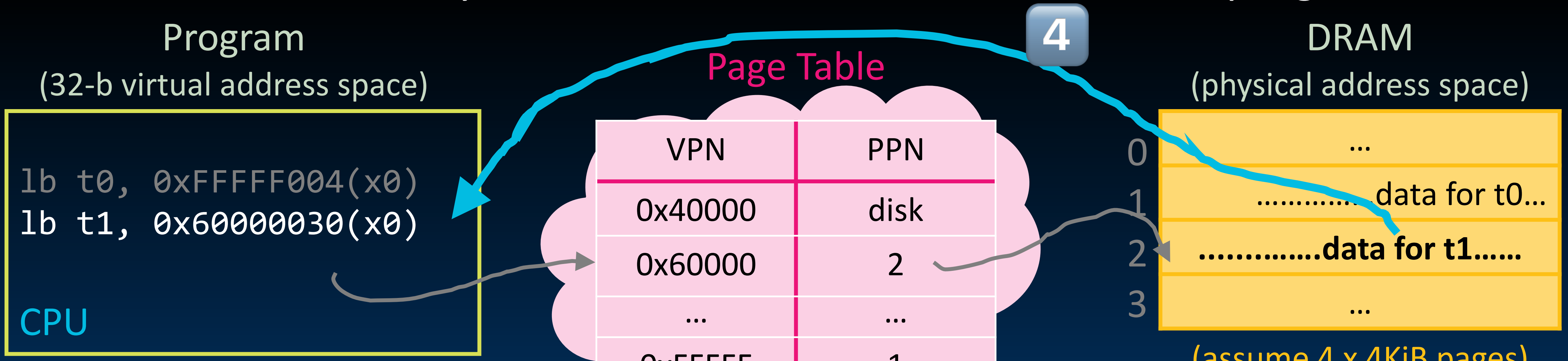
```
lb t0, 0xFFFFF004(x0)
lb t1, 0x60000030(x0)
```

VPN	PPN
0x40000	disk
0x60000	2 ✓
…	…
0xFFFFF	1

Yan, Yokota

Translation: How a Program Accesses Memory

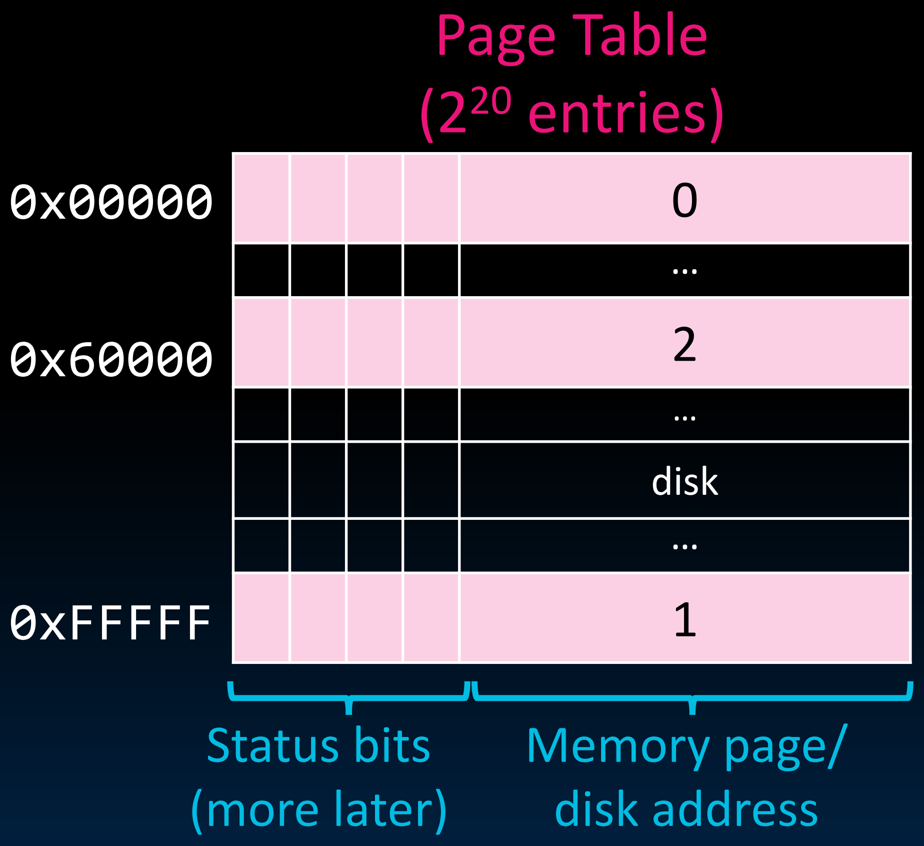
1. Program executes a load specifying a **virtual address (VA)**.
2. Computer translates **VA** to the **physical address (PA)** in memory.
   - Extract virtual page number (VPN) from VA (e.g., top 20 bits if page size 4KiB = 2^{12} B)
   - Look up physical page number (PPN) in page table
   - Construct PA: physical page number + offset (from virtual address)
3. **Page Fault**: If **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.



VPN	PPN
0x40000	disk
0x60000	2
…	…
0xFFFFF	1

What Do Page Tables Look Like?

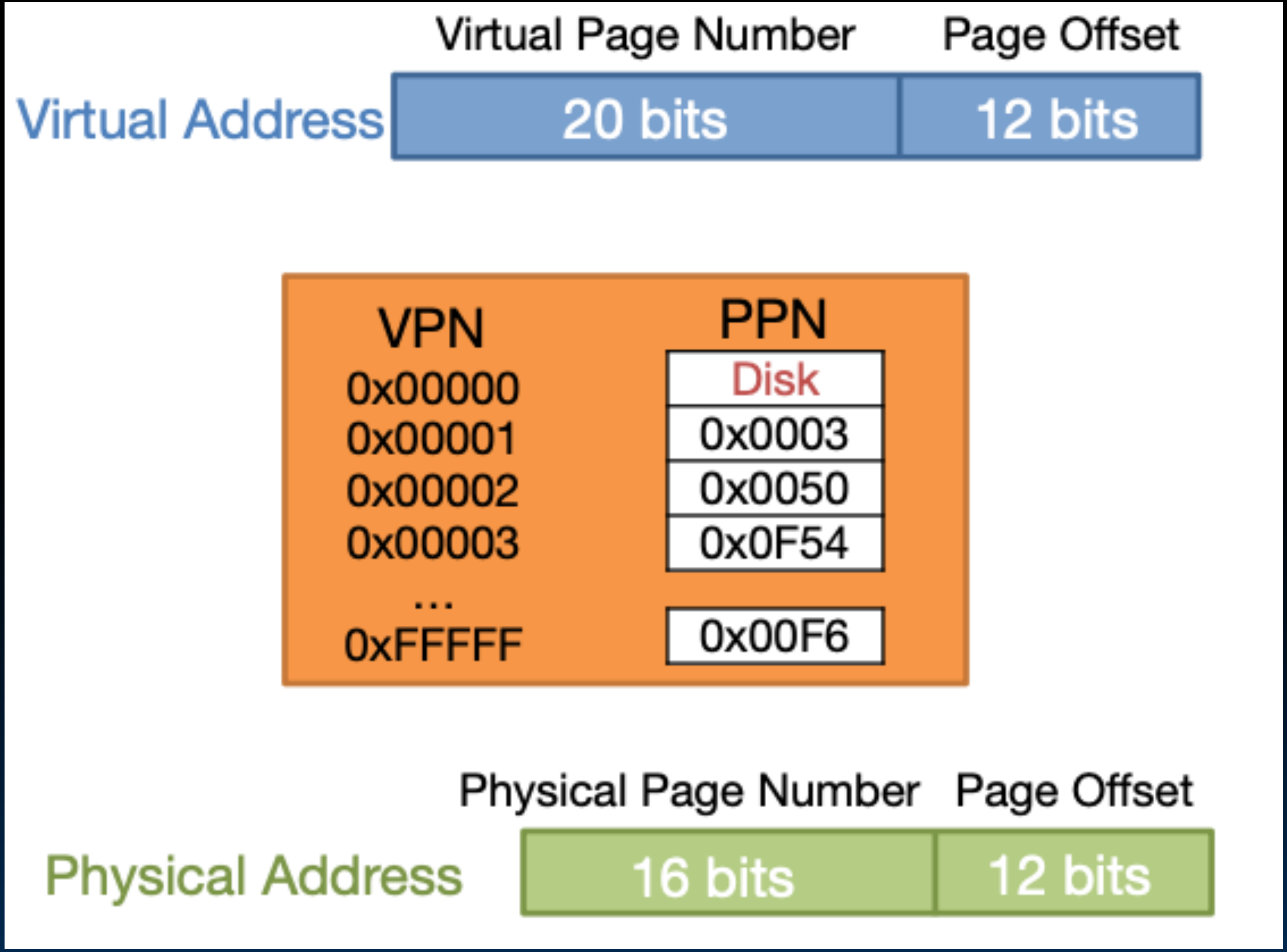
- E.g., 32-bit virtual address space, 4-KiB pages
  - 2^{32} virtual addresses / (2^{12} B/page) = 2^{20} virtual page numbers
- One **Page Table** per process:
  - One entry per virtual page number.
  - Entry has physical page number (or disk address) as well as status bits.
- Note: A Page Table is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



Agenda

Practice

Translation Example

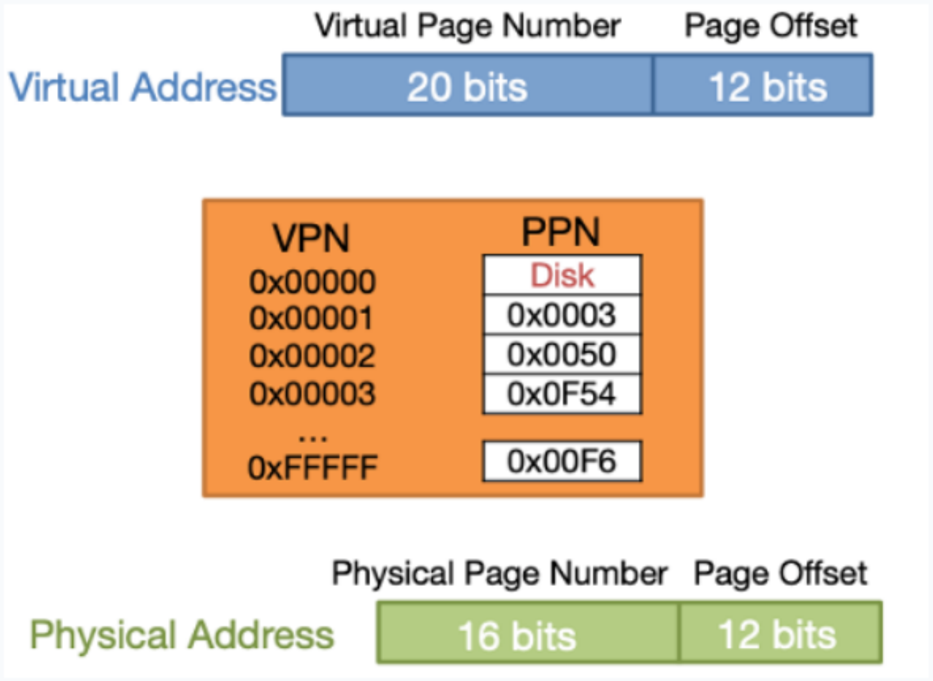


0x00003450

What Physical Address does this Virtual Address translate to?

A. 0x00003450
B. 0x0000250
C. 0x00503450
D. 0x0F543450
E. 0x0F54450
F. Disk/Other

What Physical Address does the virtual address 0x00003450 translate to?



A. 0x00003450 0%

B. 0x0000250 0%

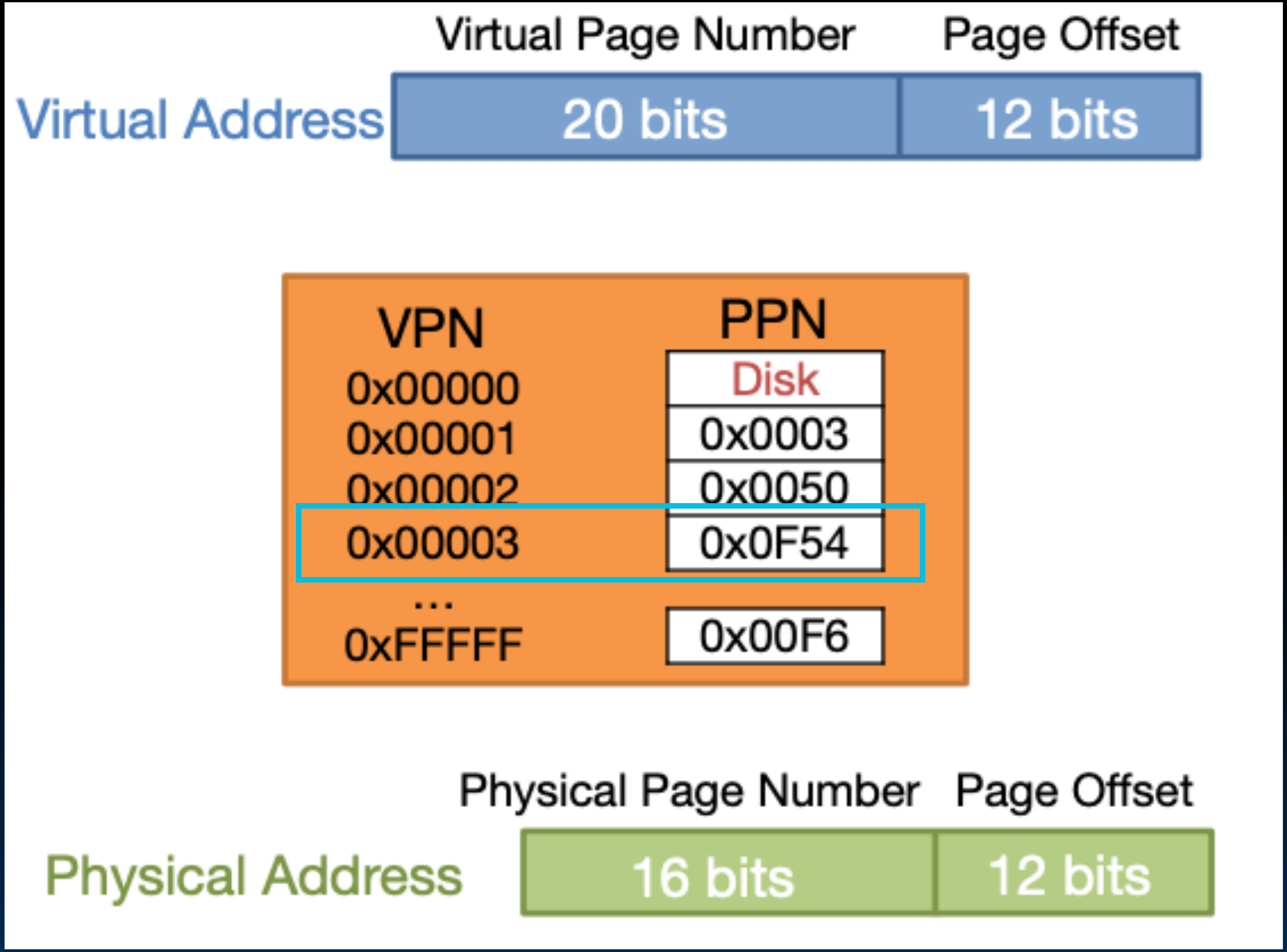
C. 0x00503450 0%

D. 0x0F543450 0%

E. 0x0F54450 0%

F. Disk/Other 0%

Translation Example



0x**00003**450

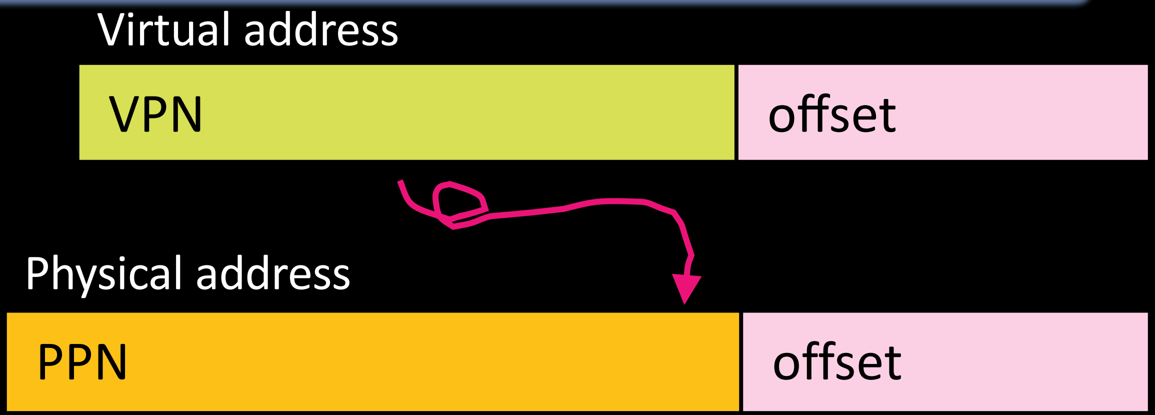
What Physical Address does this Virtual Address translate to?

A. 0x00003450
B. 0x0000250
C. 0x00503450
D. 0x0F543450
[✓ E.] 0x0F54450 0x<u>**0F54**</u>450
F. Unknown/Other

Page offset bits do not change!

Setup

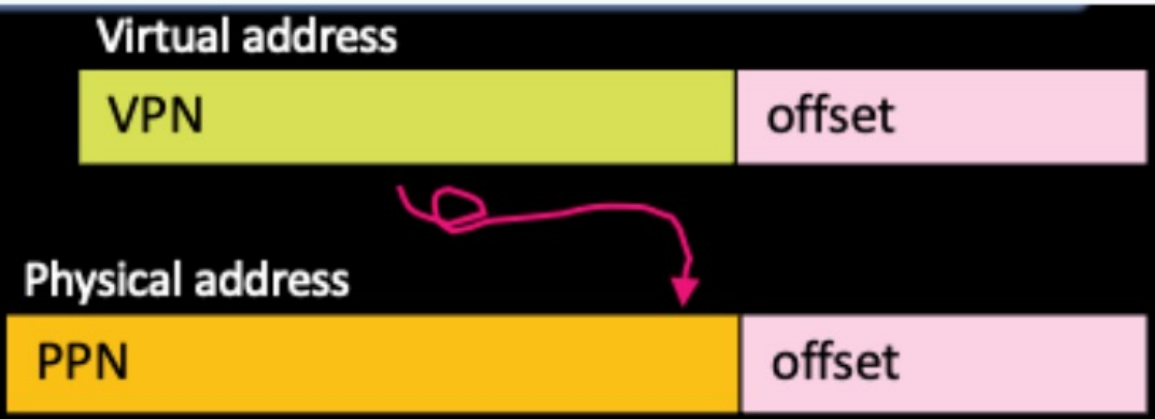
- Assume a 32-bit machine with 8GiB of RAM and 16KiB pages.
- How many bits would there be for each of the following?



	A.	B.	C.	D.	E.	F.
1. Page offset	14	15	16	19	20	Other
2. VPN	14	15	16	19	20	Other
3. PPN	14	15	16	19	20	Other

How many bits would there be for each of the following?

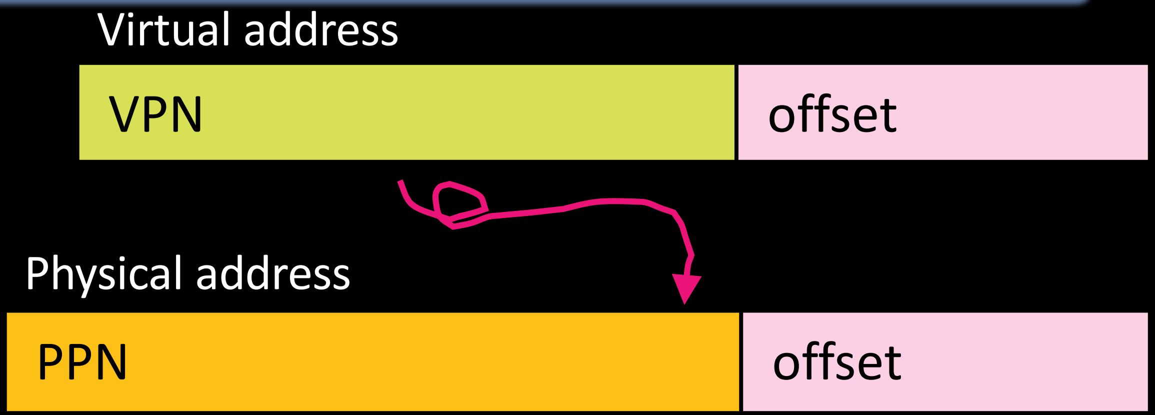
- Assume a 32-bit machine with 8GiB of RAM and 16KiB pages.
- How many bits would there be for each of the following?



		A.	B.	C.	D.	E.	F.
1.	Page offset	14	15	16	19	20	Other
2.	VPN	14	15	16	19	20	Other
3.	PPN	14	15	16	19	20	Other

Setup

- Assume a **32-bit** machine with 8GiB of RAM and 16KiB pages.
- How many bits would there be for each of the following?



1. Page offset $= \log_2(16\ \text{KiB}) = \log_2(2^4 * 2^{10}) = 14$ bits
2. VPN $= \mathbf{32} - 14 = 18$ bits
3. PPN $= \log_2(8\text{GiB}) - 14 \quad = \log_2(2^3 * 2^{30}) - 14 = 33 - 14 = 19$ bits

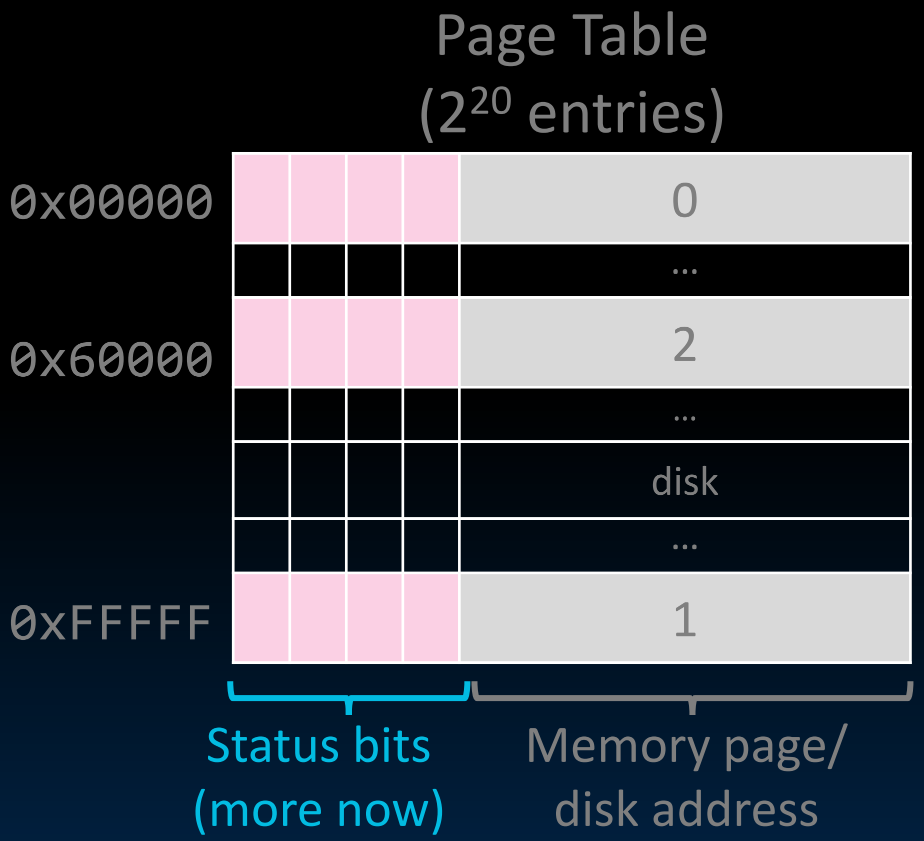
Agenda

Page Table Details I

- Virtual Memory and Virtual Addresses
- Paged Memory
- Address Translation
- Practice
- Page Table Details I

Page Table Status Bits

- E.g., 32-bit virtual address space, 4-KiB pages
  - 2^{32} virtual addresses / (2^{12} B/page) = 2^{20} virtual page numbers
- 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”



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:

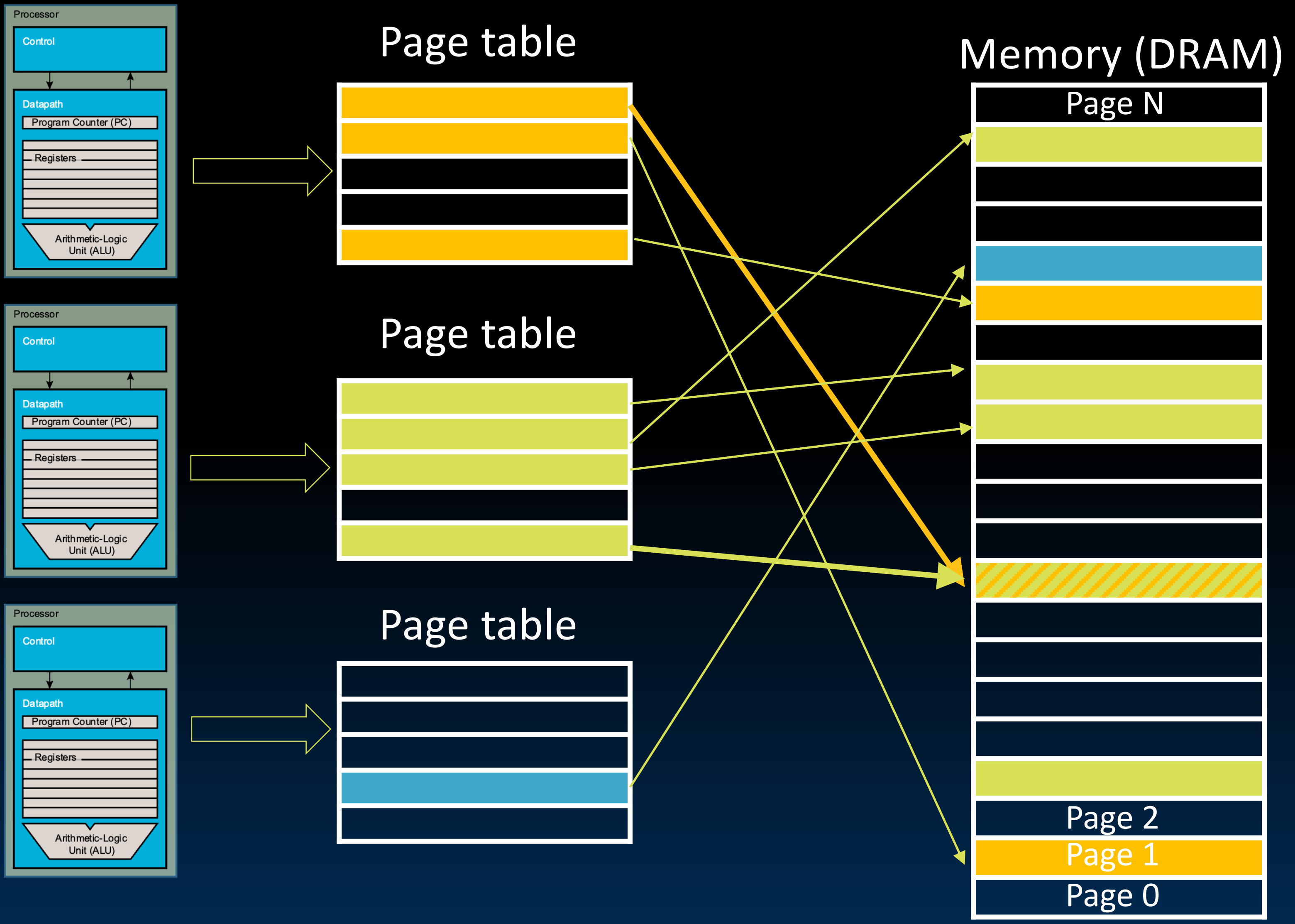
- Isolate memory between processes.
- Each process gets dedicated "private" memory.
- Errors in one program won't corrupt memory of other programs.
- Prevent user programs from messing with OS's memory.

What if process tries to modify instructions or system data?

Protection with Page Tables (1/2)

(got to here)

- 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

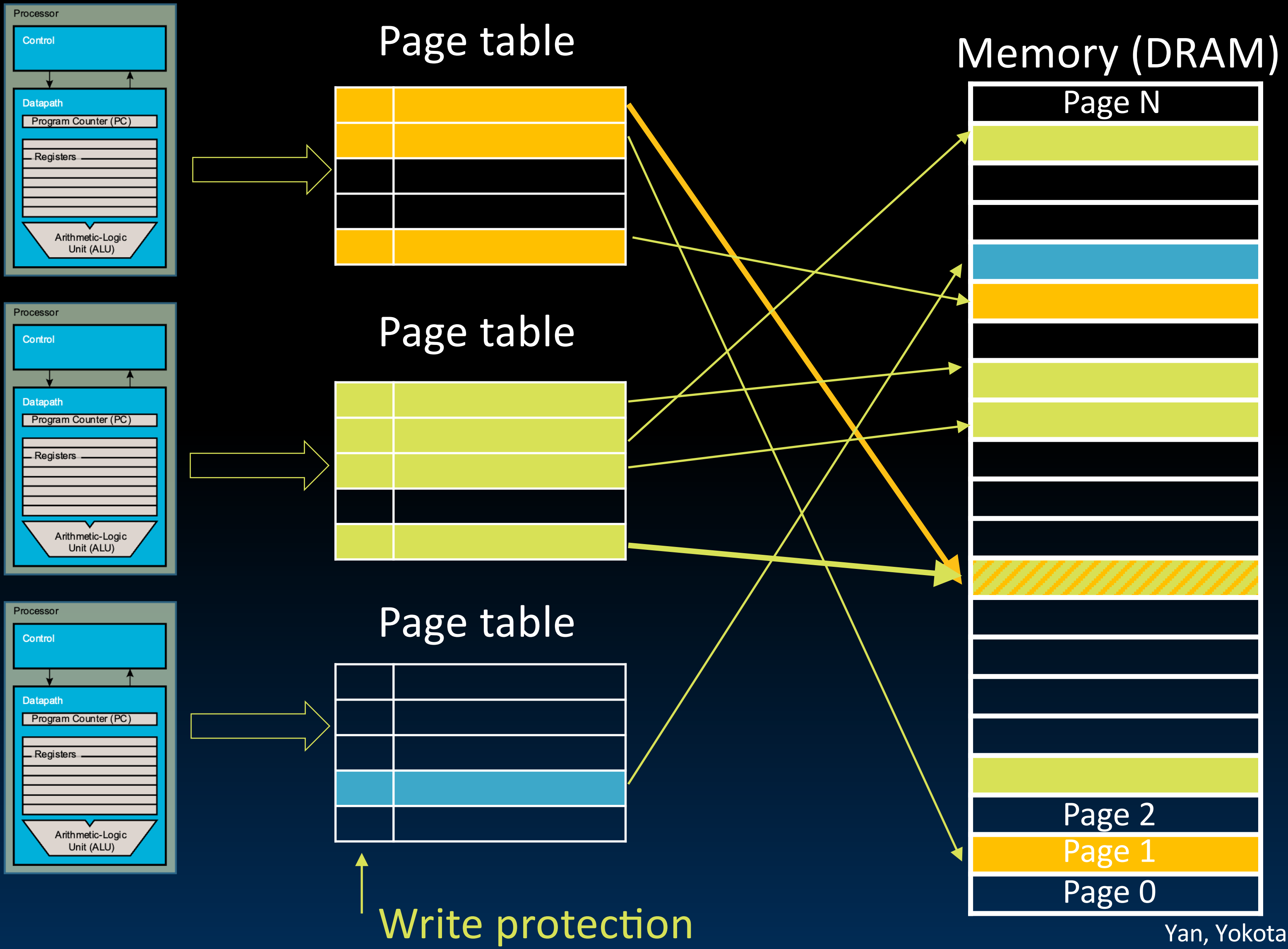


Yan, Yokota

Protection with Page Tables (2/2)

(for next time)

- 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)



Yan, Yokota

Page Tables Are Stored in Memory (1/2)

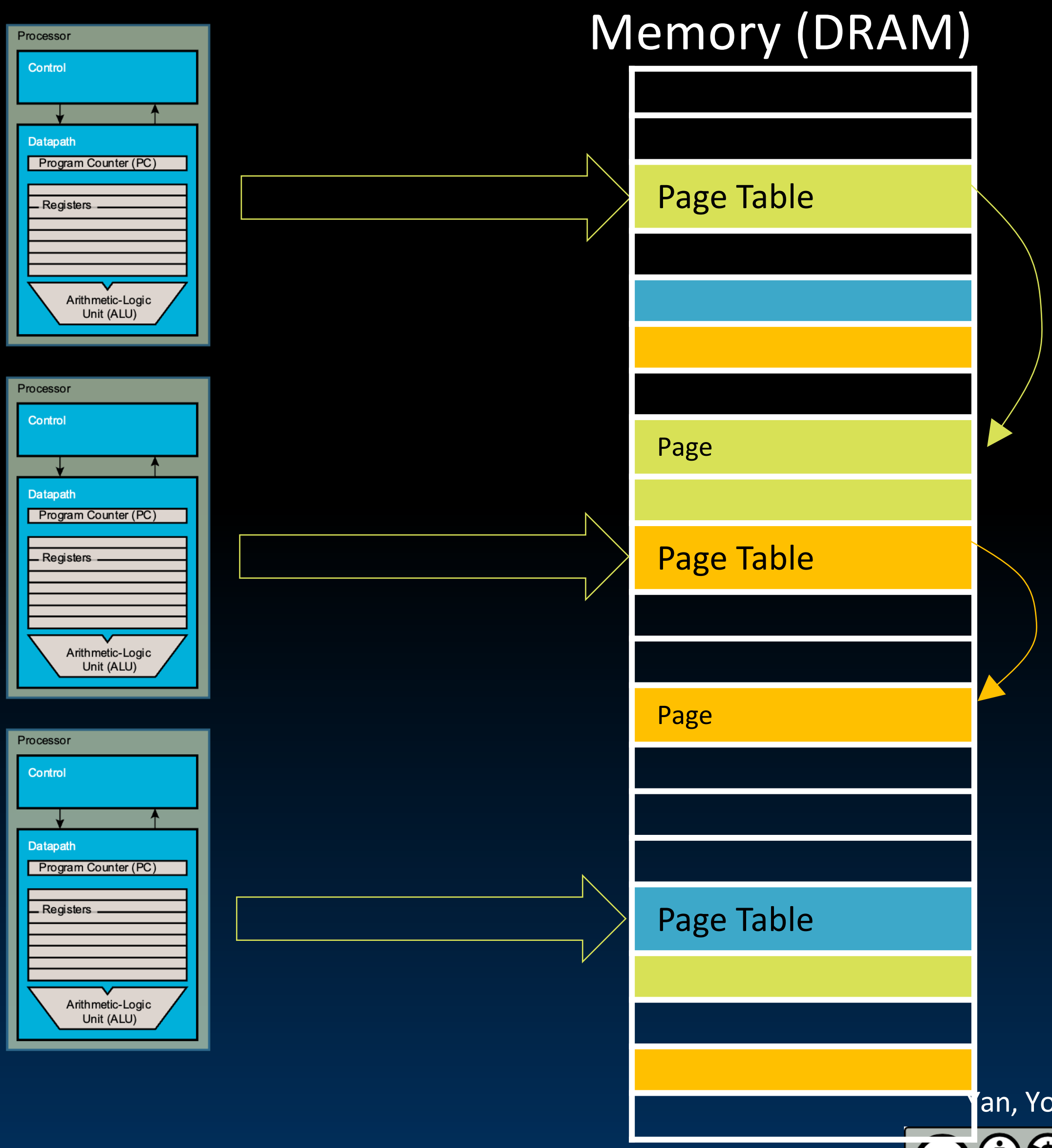
(for next time)

- E.g., 32-Bit virtual address space, 4-KiB pages
  - Single page table size (suppose each entry is (4B) + status bits):
    - 4 x 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"?
  - What if a page is not found in memory?
  - Write-back or write-through?
  - How to incorporate caches with virtual memory?