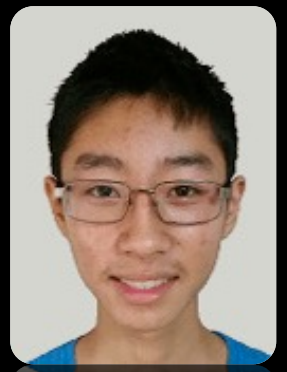
UC Berkeley
Teaching Professor
Lisa Yan

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)



UC Berkeley
Lecturer
Justin Yokota

Virtual Memory II

Virtual Machine Warmup

- Suppose we have a 32-bit virtual address space with 16GiB DRAM and 4KiB pages. True or False?

1. There are $2^{32} / 2^{12} = \mathbf{2^{20}}$ Virtual Page Numbers. — 1. True False
2. There are $2^{4}2^{30} / 2^{12} = \mathbf{2^{22}}$ Physical Page Numbers. — 2. True False
3. Virtual addresses are 32-bits, where the bottom **12 bits** are used to reference page offset. — 3. True False
4. Both the Virtual Page Size and the Physical Page Size are **4 KiB**. — 4. True False
5. **# page table entries** = # virtual page numbers. — 5. True False
6. If 2^{20} page table entries, where each entry is 4 B (to store PPN + status bits), then the total **page table size is 4 MiB**. — 6. True False
7. Page tables are stored in memory. — 7. True False
8. Assuming no caches, then each `lw` (or `sw`) requires **2 memory accesses**. — 8. True False

Yan, Yokota

[Warmup] Select T/F for each question.

Virtual Machine Warmup

- Suppose we have a 32-bit virtual address space with 16GiB DRAM and 4KiB pages. True or False?

1. There are $2^{32} / 2^{12} = \mathbf{2^{20}}$ Virtual Page Numbers. — 1. True False
2. There are $2^4 2^{30} / 2^{12} = \mathbf{2^{22}}$ Physical Page Numbers. — 2. True False
3. Virtual addresses are 32-bits, where the bottom **12 bits** are used to reference page offset. — 3. True False
4. Both the Virtual Page Size and the Physical Page Size are **4 KiB**. — 4. True False
5. **# page table entries** = # virtual page numbers. — 5. True False
6. If 2^{20} page table entries, where each entry is 4 B (to store PPN + status bits), then the total **page table size is 4 MiB**. — 6. True False
7. Page tables are stored in memory. — 7. True False
8. Assuming no caches, then each `lw` (or `sw`) requires **2 memory** — 8. True False

Virtual Machine Warmup

- Suppose we have a 32-bit virtual address space with 16GiB DRAM and 4KiB pages. True or False?

1. There are $2^{32} / 2^{12} = \mathbf{2^{20}}$ Virtual Page Numbers. — 1. True False
2. There are $2^4 2^{30} / 2^{12} = \mathbf{2^{22}}$ Physical Page Numbers. — 2. True False
3. Virtual addresses are 32-bits, where the bottom **12 bits** are used to reference page offset. — 3. True False
4. Both the Virtual Page Size and the Physical Page Size are **4 KiB**. — 4. True False
5. **# page table entries** = # virtual page numbers. — 5. True False
6. If 2^{20} page table entries, where each entry is 4 B (to store PPN + status bits), then the total **page table size is 4 MiB**. — 6. True False
7. Page tables are stored in memory. — 7. True False
8. Assuming no caches, then each `lw` (or `sw`) requires **2 memory accesses**. — 8. True False

Yan, Yokota

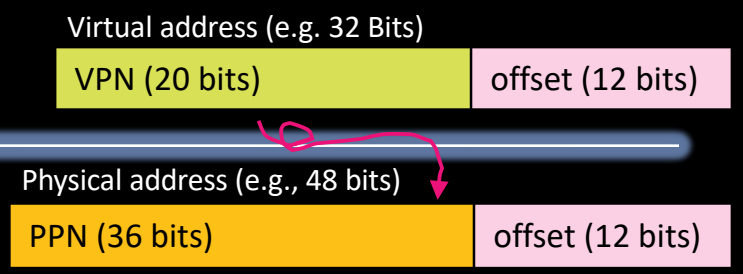
Virtual Machine Warmup

- Suppose we have a 32-bit virtual address space with 16GiB DRAM and 4KiB pages. True or False?

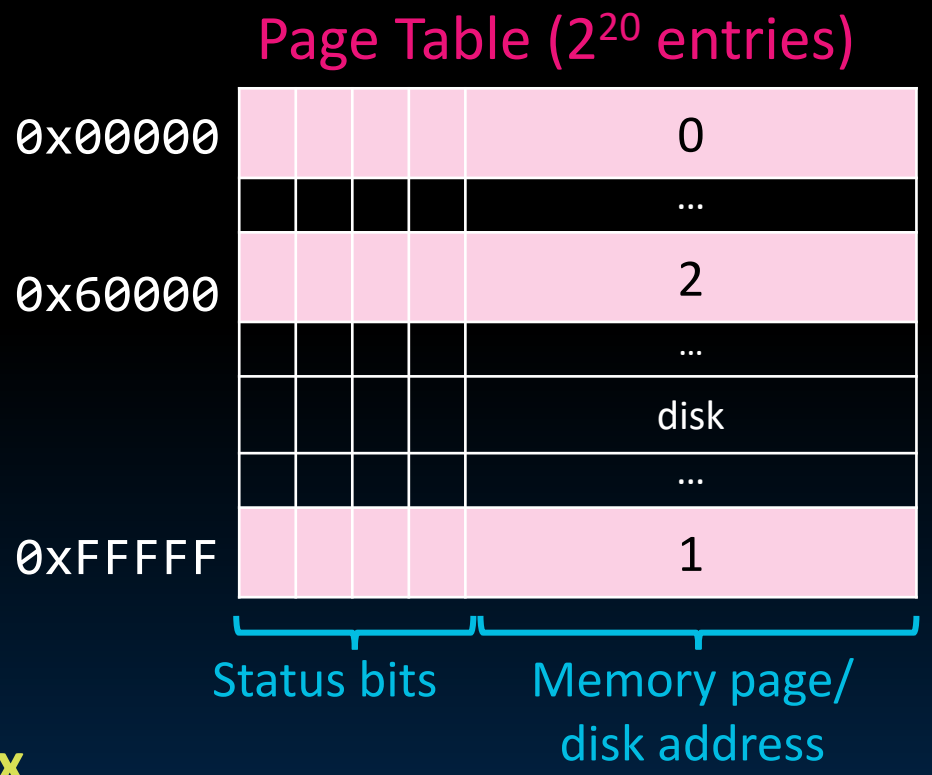
1. There are $2^{32} / 2^{12} = \mathbf{2^{20}}$ Virtual Page Numbers. — 1. True False
2. There are $2^{4}2^{30} / 2^{12} = \mathbf{2^{22}}$ Physical Page Numbers. — 2. True False
3. Virtual addresses are 32-bits, where the bottom **12 bits** are used to reference page offset. — 3. True False
4. Both the Virtual Page Size and the Physical Page Size are **4 KiB**. — 4. True False
5. **# page table entries** = # virtual page numbers. — 5. True False
6. If 2^{20} page table entries, where each entry is 4 B (to store PPN + status bits), then the total **page table size is 4 MiB**. — 6. True False
7. Page tables are stored in memory. — 7. True False
8. Assuming no caches, then each `lw` (or `sw`) requires **2 memory accesses**. — 8. True False

Yan, Yokota

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.
  - The VPN effectively functions as the page table **index**.



Yan, Yokota

Virtual Machine Warmup

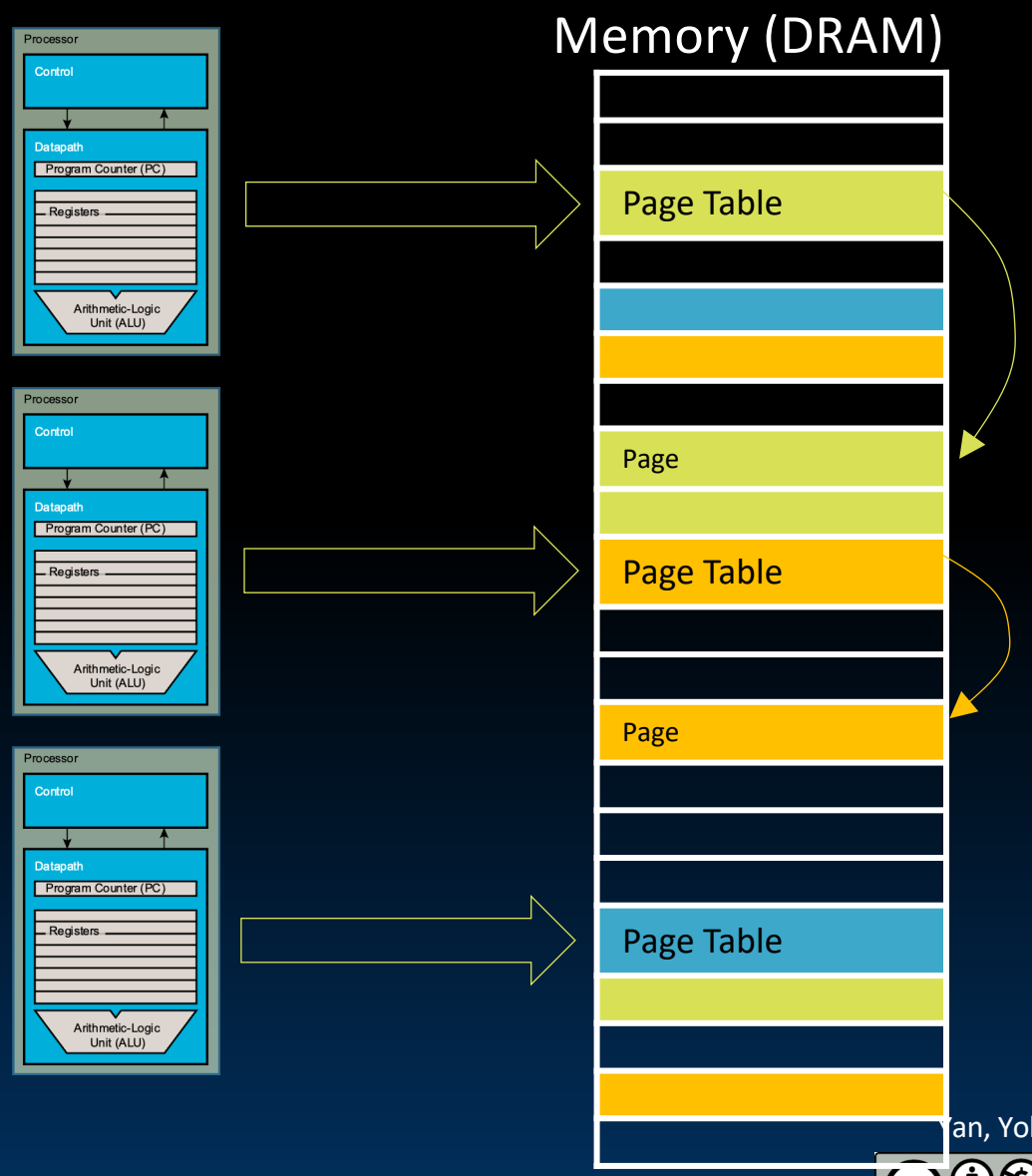
- Suppose we have a 32-bit virtual address space with 16GiB DRAM and 4KiB pages. True or False?

1. There are $2^{32} / 2^{12} = \mathbf{2^{20}}$ Virtual Page Numbers. — 1. True False
2. There are $2^4 2^{30} / 2^{12} = \mathbf{2^{22}}$ Physical Page Numbers. — 2. True False
3. Virtual addresses are 32-bits, where the bottom **12 bits** are used to reference page offset. — 3. True False
4. Both the Virtual Page Size and the Physical Page Size are **4 KiB**. — 4. True False
5. **# page table entries** = # virtual page numbers. — 5. True False
6. If 2^{20} page table entries, where each entry is 4 B (to store PPN + status bits), then the total **page table size is 4 MiB**. — 6. True False
7. Page tables are stored in memory. — 7. True False
8. Assuming no caches, then each `lw` (or `sw`) requires **2 memory accesses**. — 8. True False

Yan, Yokota

Page Tables Are Stored in Memory…

- Reasoning:
  - A 4 MiB page table is 0.02% of 16 GiB DRAM, but is much too large for a cache.
- Caveat: `lw/sw` then requires two memory accesses.
  - (1) Read page table (stored in main memory) to translate to physical address.
  - (2) Read physical page, also in main memory.
- To minimize the performance penalty:
  - Transfer blocks (not entire pages) between DRAM and processor cache.
  - Use a cache for frequently used page table entries… (more later, TLB)



Agenda

Page Table Details II: Page Faults, Write Policy

- Page Table Details: Page Faults, Write Policy
- OS: Trap Handler
- [Review] Caches vs. Virtual Memory
- Translation Lookaside Buffer

Yan, Yokota

Page Faults

- Page table entries store status to indicate if the page is in memory (DRAM) or only on disk.
  - On each memory access, check the page table entry "*valid*" status bit.
- Valid → In DRAM
  - Read/write data in DRAM
- Not Valid → On disk
  - Triggers a *Page Fault*; OS intervenes to allocate the page into DRAM.
  - If out of memory, first evict a page from DRAM.
  - Store evicted page to disk.
  - Read requested page from disk into DRAM.
  - Finally, read/write data in DRAM.

The page replacement policy is usually done in OS/software; this overheard << disk access time. (usually an LRU approximation; more in Ch 5.7)

Yan, Yokota

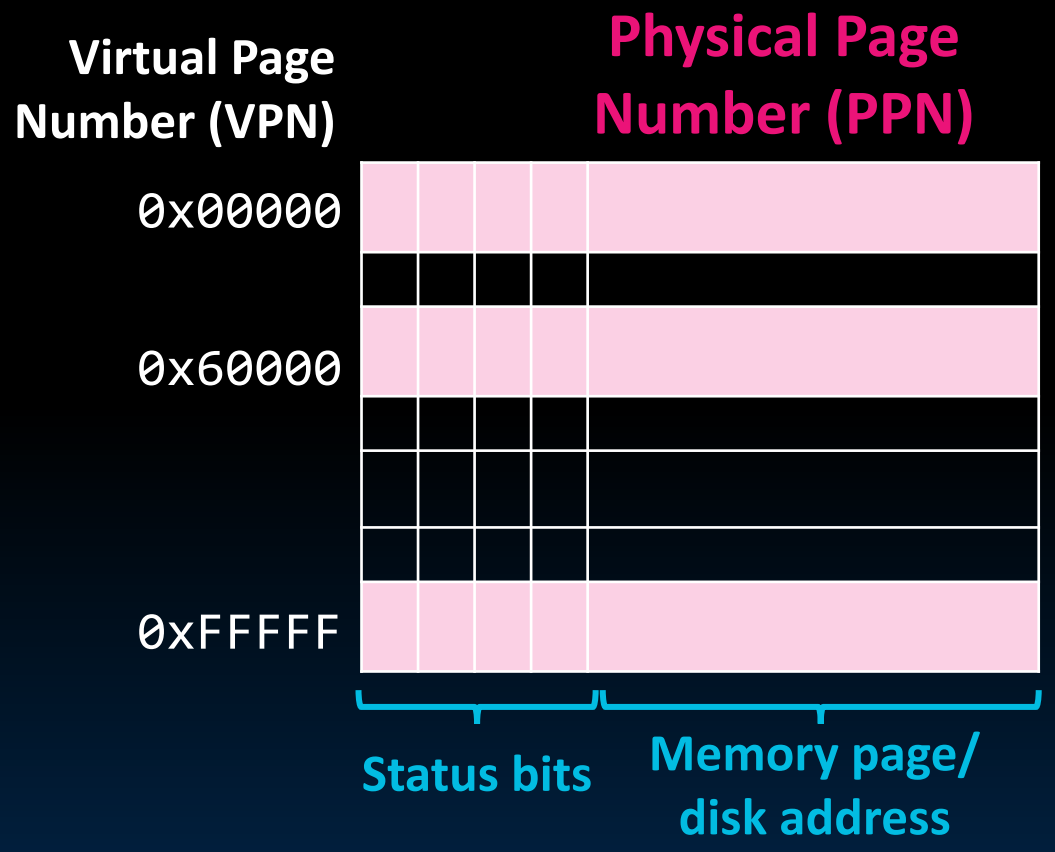
Memory's Write Policy?

- DRAM acts like a "cache" for disk.
  - Should writes always go directly to disk (write-through), or
  - Should writes only go to disk when page is evicted (write-back)?
- Answer: All virtual memory systems use **write-back**.
  - Disk accesses take too long!

Yan, Yokota

Summary of Page Table Status Bits

- Valid Bit
  - On: Page is in RAM
  - (Off: Page is on disk → Page Fault)
- Dirty Bit
  - On: Page in RAM is more up-to-date than page on disk
- Write Protection Bit
  - On: If process writes to page, trigger exception
  - (Example: if page is program code, system data, etc.)



Yan, Yokota

(dramatic pause)

Yan, Yokota

Q: How does the OS *manage* Virtual Memory (e.g., page faults)?

A: Exceptions!

Yan, Yokota

Agenda

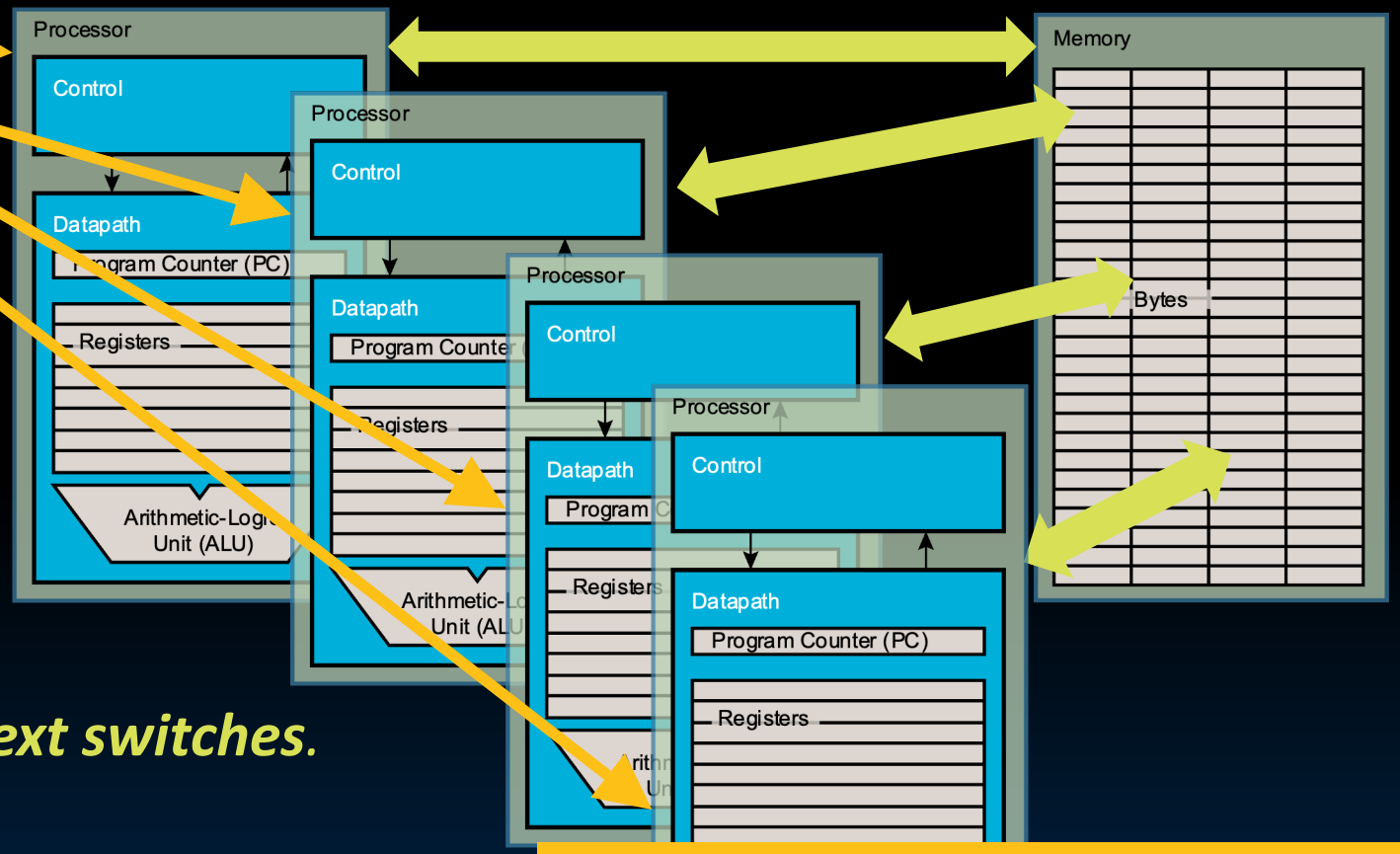
OS: Trap Handler

- Page Table Details: Page Faults, Write Policy
- OS: Trap Handler
- [Review] Caches vs. Virtual Memory
- Translation Lookaside Buffer

Yan, Yokota

Memory Is Shared

```
0:04.34 /usr/libexec/UserEventAgent (Aqua)
0:10.60 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/cfprefsd agent
0:04.71 /usr/sbin/usernoted
0:02.35 /usr/libexec/nsurlsessiond
0:28.68 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameSe
0:01.90 /System/Library/CoreServices/cloudphotos
0:49.72 /usr/libexec/secinitd
0:01.66 /System/Library/PrivateFrameworks/TCC.fr
0:12.68 /System/Library/Frameworks/Accounts.fram
0:09.56 /usr/libexec/SafariCloudHistoryPushAgent
0:00.27 /System/Library/PrivateFrameworks/CallHi
0:00.74 /System/Library/CoreServices/mapspushd
0:00.79 /usr/libexec/fmfd
```



- 100's of processes
  - OS multiplexes these over available cores.
  - If single-core, performs ***context switches***.
- But what about memory?
  - There is only one! DRAM
  - Single-core: An OS context switch cannot just "save" memory's contents…too costly!

Context switches are where the OS switches out the state of the processor between processes (i.e., programs) through use of the **trap handler**.

Exceptions and Interrupts

- Exceptions
  - Caused by an event *during* the execution of the current program.
  - *Synchronous*; must be handled immediately.
  - Examples:
    - Illegal instruction
    - Divide by zero
    - *Page fault*
    - Write protection violation
- Interrupts
  - Caused by an event *external* to the current running program.
  - *Asynchronous* to current program; does not need to be handled immediately (but should be soon).
  - Examples:
    - Key press
    - Disk I/O

Yan, Yokota

Traps Handle Exceptions and Interrupts

- The **trap handler** is code that services exceptions and interrupts.

1. Complete all instructions before the faulting instruction.
2. Flush all instructions after the faulting instruction.
   - Like pipeline hazard: convert to noops/"bubbles."
   - Also flush faulting instruction.
3. Transfer execution to trap handler (runs in **supervisor mode**).
   - Optionally return to original program and re-execute instruction.

If the trap handler returns, then from the program's point of view it must look like nothing has happened!

Yan, Yokota

Supervisor Mode vs. User Mode

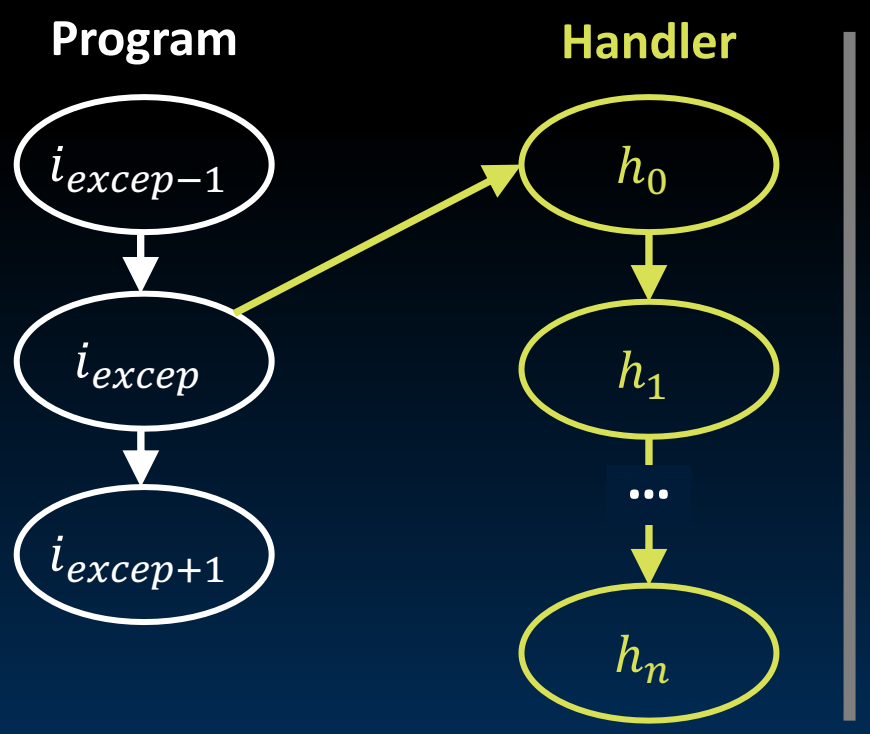
[extra]

- If an application goes wrong (or rogue, e.g., malware), it could crash the entire machine!
- CPUs have a hardware **supervisor mode** (i.e., **kernel mode**).
  - Set by a status bit in a special register.
  - An OS process in supervisor mode helps enforce constraints to other processes, e.g., access to memory, devices, etc.
  - Supervisor mode is a bit like "superuser"…
    - Errors in supervisory mode are often catastrophic (blue "screen of death", or "I just corrupted your disk").
- By contrast, in **user mode**, a process can only access a subset of instructions and (physical) memory.
  - Can change *out* of supervisor mode using a special instruction (e.g. `sret`).
  - Cannot change *into* supervisor mode directly; instead, HW interrupt/exception.
  - The OS mostly runs in user mode! Supervisor mode is used sparingly.

Yan, Yokota

The Trap Handler

1. Save the state of the current program.
   - Save ALL of the registers!
2. Determine what caused the exception/interrupt.
3. Handle exception/interrupt…



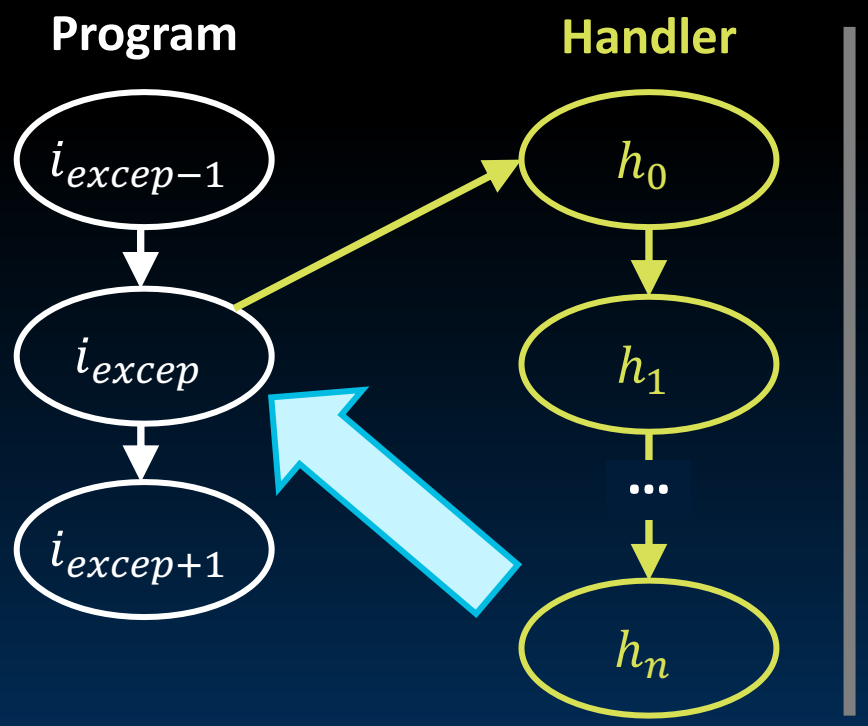
Yan, Yokota

The Trap Handler

1. Save the state of the current program.
   - Save ALL of the registers!
2. Determine what caused the exception/interrupt.
3. Handle exception/interrupt…then do one of two things:

Continue execution of the program:

4. Restore program state.
5. Return control to the program.



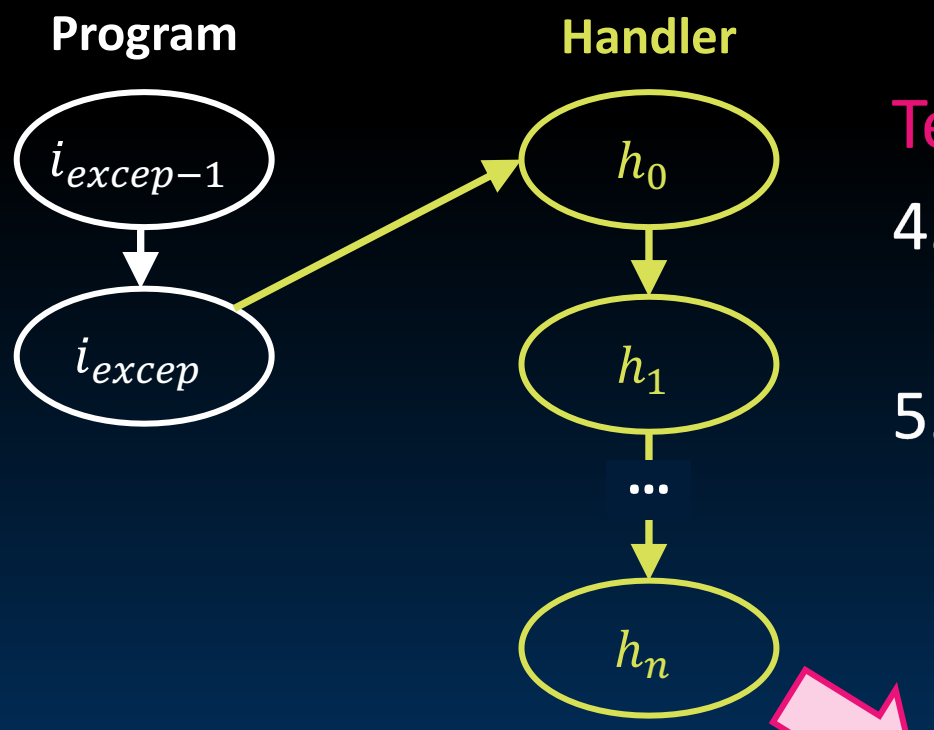
Yan, Yokota

The Trap Handler

1. Save the state of the current program.
   - Save ALL of the registers!
2. Determine what caused the exception/interrupt.
3. Handle exception/interrupt…then do one of two things:

Continue execution of the program:

4. Restore program state.
5. Return control to the program.



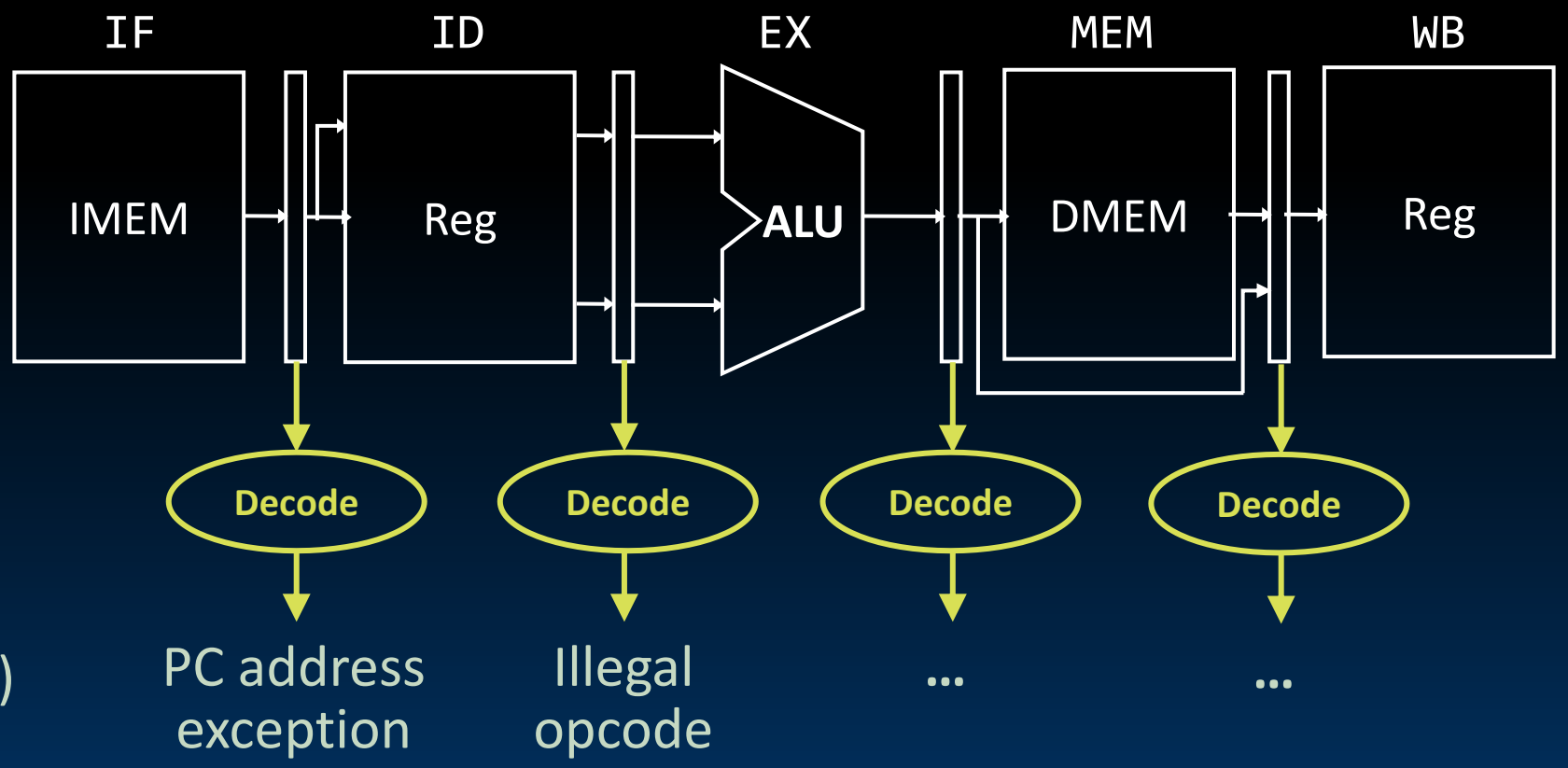
Terminate the program:

4. Free the program resources, etc.
5. Schedule a new program.

Yan, Yokota

Exceptions in a 5-Stage Pipeline

- **Traps** are handled similarly to pipeline hazards.
- In RISC-V, the exception cause can be inferred by the **faulting instruction** and its **current pipeline stage**.



Yan, Yokota

Handling Context Switches

- Once more, the **context switch**:
  - OS switches between processes (i.e., programs) by changing the internal state of the processor.
  - Allows a single processor to "simultaneously" run many programs.
- At a high-level:
  - The OS sets a timer. When it expires, perform a ***hardware interrupt***.
  - Trap handler saves all register values, including:
    - Program Counter (PC)
    - ***Page Table Register*** (SPTBR in RV32I)
      - The memory *address* of the active process's page table.
  - Trap handler then loads in the next process's registers and returns to user mode.

Yan, Yokota

Handling Page Faults

- Recall **page faults**:
  - An accessed page table entry has valid bit off → data is not in DRAM.
- Page faults are handled by the trap handler.
  - The *page fault exception handler* initiates transfers to/from disk and performs any page table updates.
  - (If pages needs to be swapped from disk, perform *context switch* so that another process can use the CPU in the meantime.)
    - (ideally need a "precise trap" so that resuming a process is easy.)
  - Following the page fault, *re-execute the instruction*.
- Side note: Write protection violations also trigger exceptions.

Yan, Yokota

Agenda

Review on your own time!

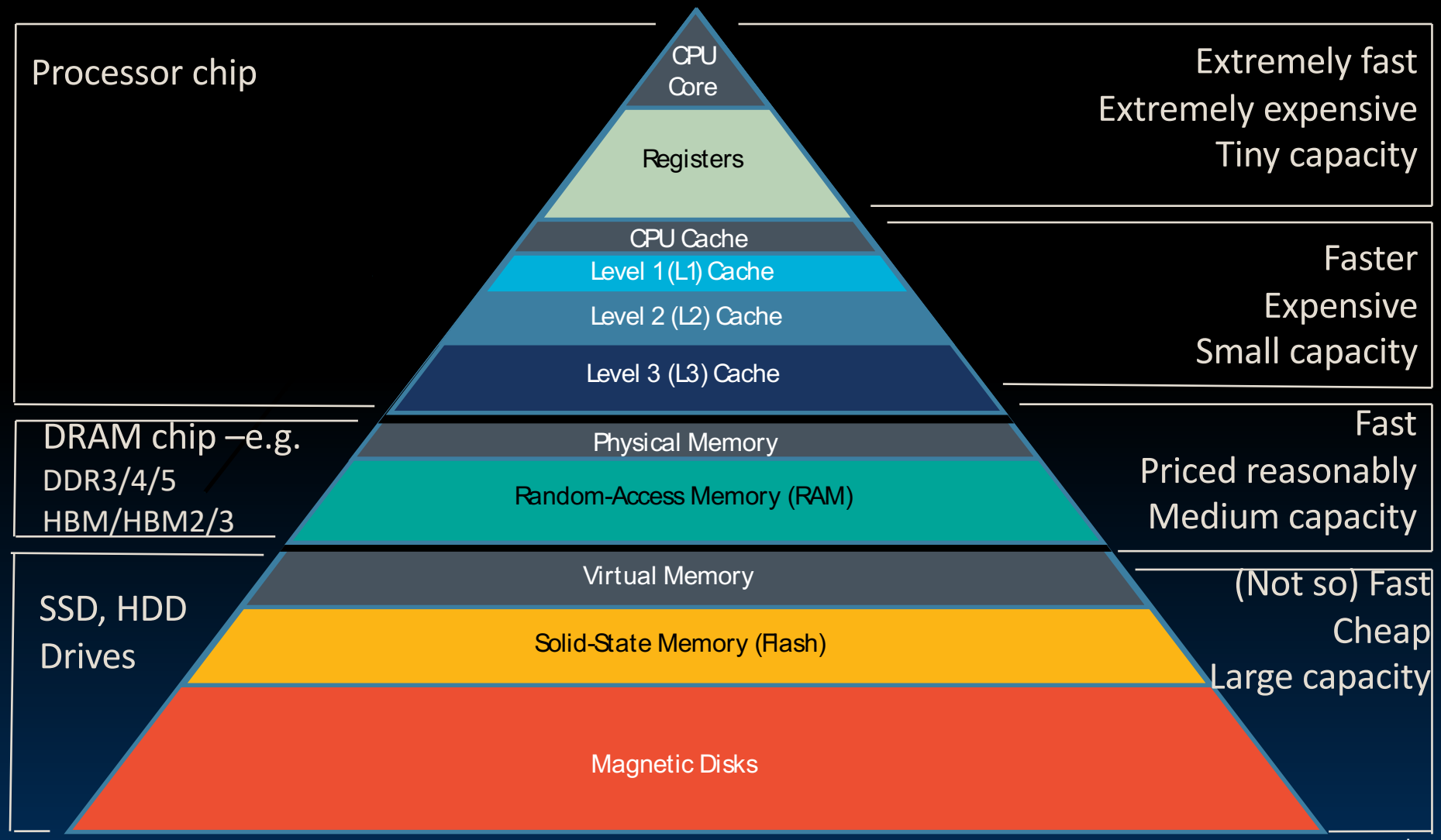
[Review] Caches vs. Virtual Memory

- Page Table Details: Page Faults, Write Policy
- OS: Trap Handler
- [Review] Caches vs. Virtual Memory
- Translation Lookaside Buffer

Yan, Yokota

The Entire Modern Memory Hierarchy

Let's review the concepts of caches and memory.



Yan, Yokota

Caches vs. Primary Memory

- Blocks, pages, (bytes, words) are all units of memory.
  - Caches: blocks
    - On modern systems, ~64B.
  - Memory: pages
    - On modern systems, ~4KiB.

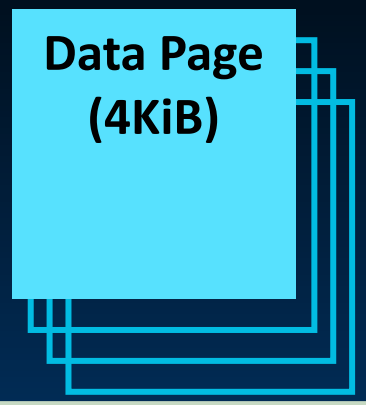
L1 Cache

V	Tag	Data
		Block (64B)
…	…	…

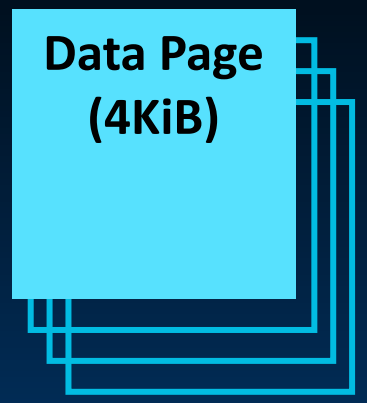
L2 Cache

V	Tag	Data
		Block (64B)
…	…	…

DRAM (primary memory)



Disk (secondary memory)



Yan, Yokota

Caches vs. Page Tables

"Cache" Paradigm: Data at each level is a **quick-access copy** of data at a lower level in the memory hierarchy.

L1 Cache

V	Tag	Data
		🙂
…	…	…

L2 Cache

V	Tag	Data
…	…	…
		🙂

DRAM (primary memory)

Data Page 🙂

Disk (secondary memory)

Data Page 🙂

Similarly, DRAM data pages are "cached" disk pages.

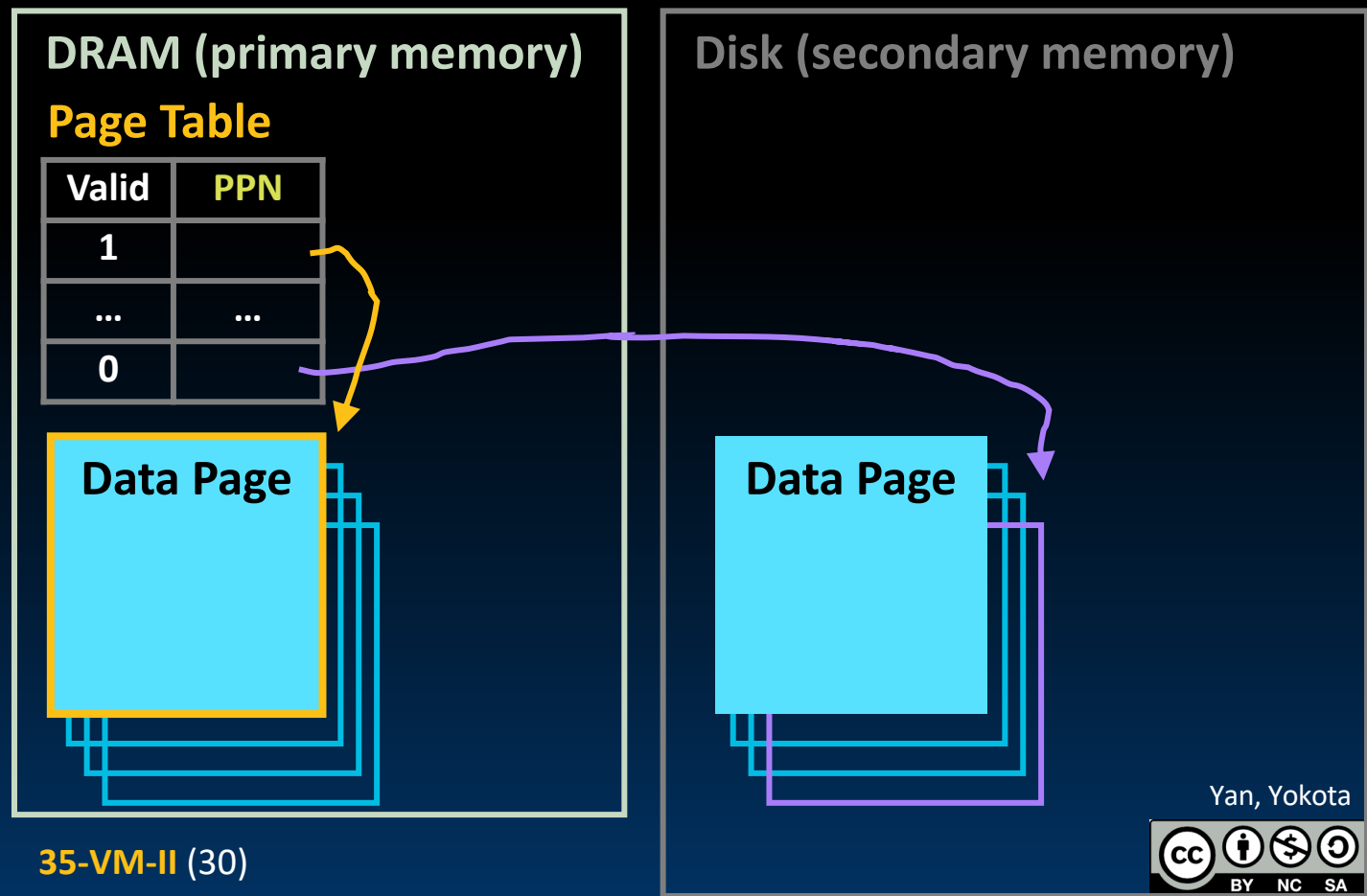
Yan, Yokota

Caches vs. Page Tables

- A **Page Table** translates addresses.
  - Page tables store physical page numbers, *not data*.

- **Page tables facilitate Demand Paging.**
  - Cache data pages in memory.
  - Access disk pages only when needed by the process.
  - Page Table keeps track of page status/location.

(assuming single-level page tables)



Yan, Yokota

Caching vs. Demand Paging

	Caching	Demand Paging
▪ Memory Unit	Block	Page
▪ Size	32B to 64B	4KiB to 8KiB
▪ Miss	Cache Miss	Page Fault
▪ Associativity	Direct-mapped, N-way Set associative, or fully associative	Fully associative (i.e., disk pages can be placed anywhere in DRAM)
▪ Replacement policy	Least Recently Used (LRU) or random	LRU (most common), or FIFO, or random
▪ Write policy	Write-through or write-back	Write-back

Yan, Yokota

Agenda

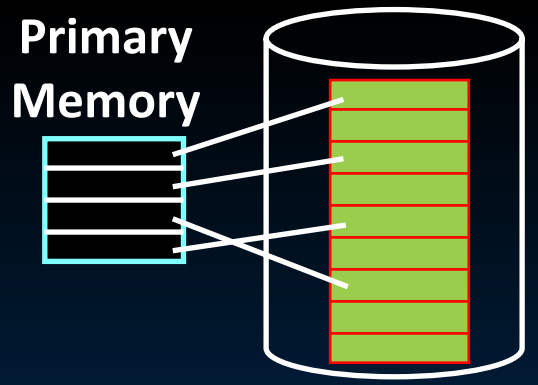
Translation Lookaside Buffer

- Page Table Details: Page Faults, Write Policy
- OS: Trap Handler
- [Review] Caches vs. Virtual Memory
- Translation Lookaside Buffer

Yan, Yokota

Modern Virtual Memory Systems

- Modern Virtual Memory Systems use address translation to provide the illusion of a large, **private**, and **uniform** storage.
  1. Privacy means **Protection**:
     - Several users/processes, each with their own private address space.
  2. Uniform storage means **Demand Paging**:
     - The ability to run programs larger than primary memory (DRAM).
     - Hides difference in machine configurations.

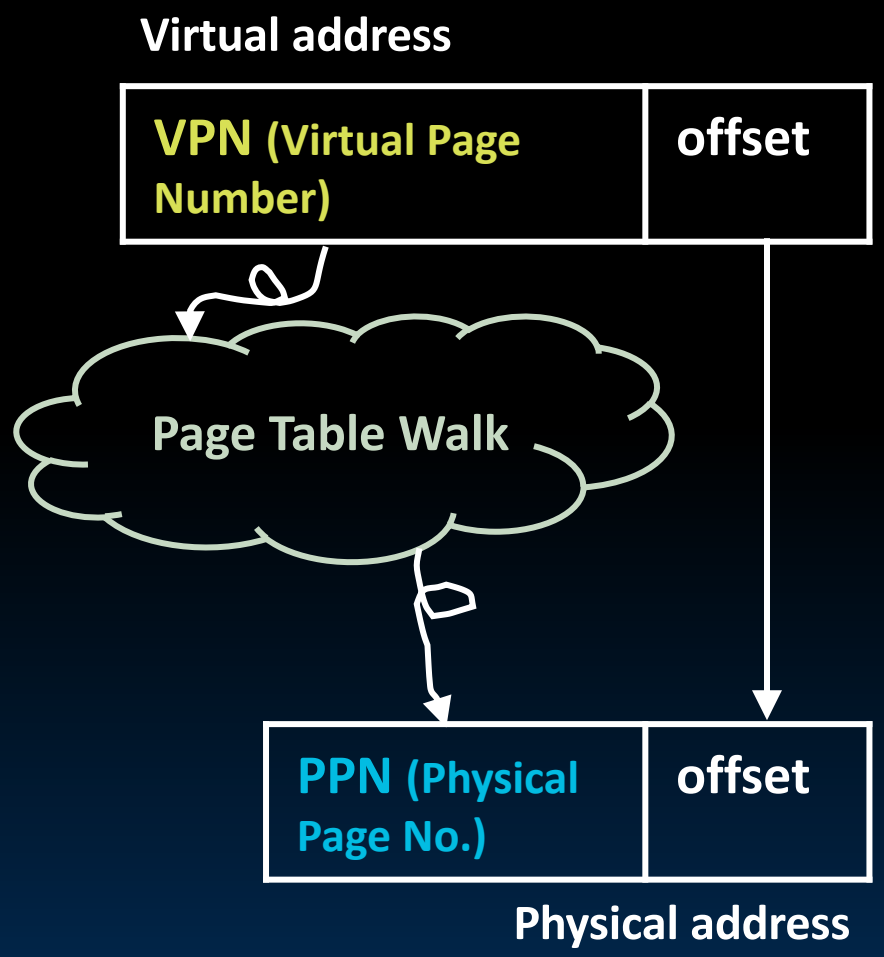


- Price: Address translation on each memory reference.

If pages tables are only stored in memory, AMAT (average memory access time) significantly increases!

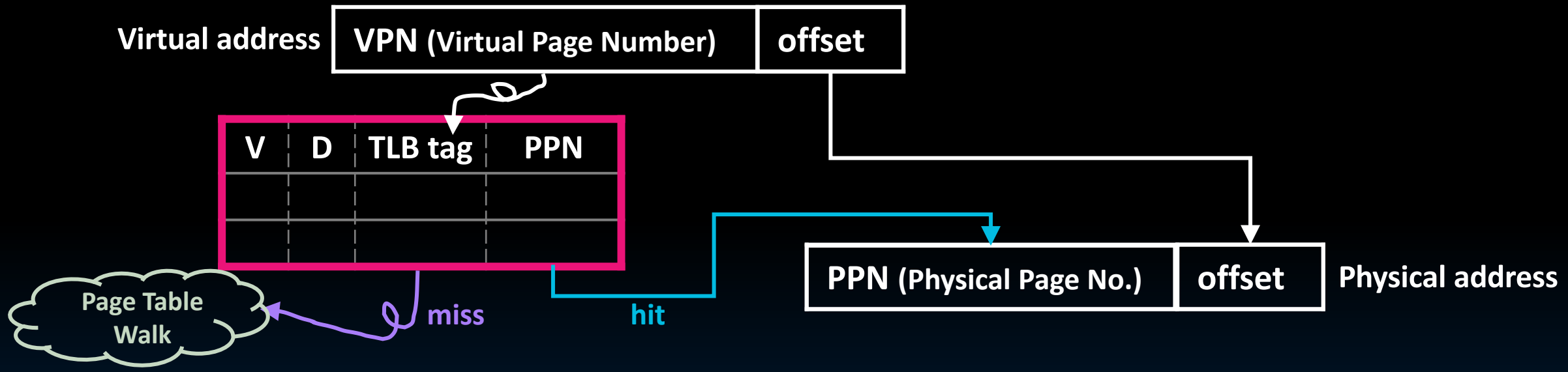
Speeding Up Address Translation

- Good Virtual Memory design should be **fast** (~1 clock cycle) and **space efficient**.
  - Every instruction/data access needs address translation.
- But if page tables are in memory, then we must perform a **page table walk** per instruction/data access:
  - Single-level page table: 2 memory accesses.
  - Two-level page table: 3 memory accesses.
- Solution: **Cache** some translations in the **Translation Lookaside Buffer (TLB)**.



The TLB Is a Cache for Address Translations

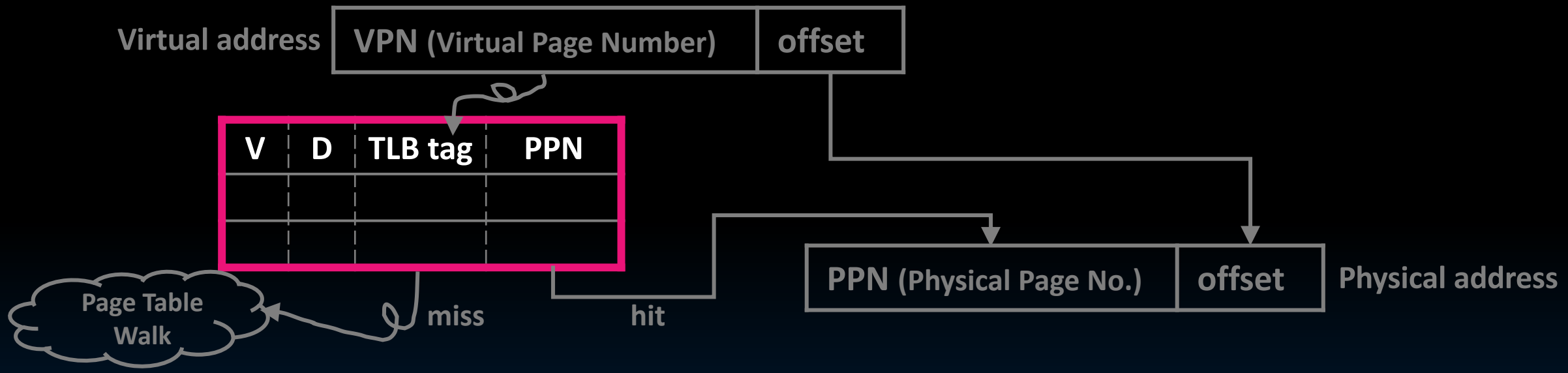
- **The Translation Lookaside Buffer (TLB) caches page table entries.**
- TLB **hit**: → Single-cycle translation ✅
- TLB **miss**: → Page table walk to refill.



Yan, Yokota

The TLB Is a Cache for Address Translations

- **The Translation Lookaside Buffer (TLB) caches page table entries.**
- TLB **hit**: → Single-cycle translation ✅
- TLB **miss**: → Page table walk to refill.



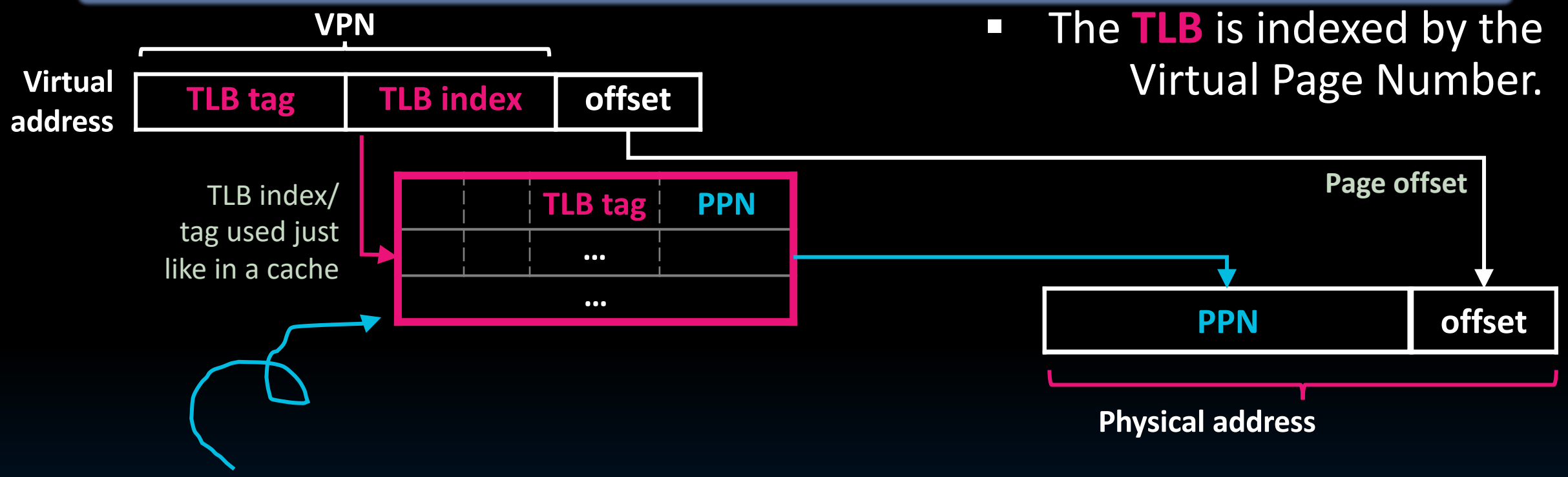
- **TLB Reach**: Size of largest virtual address space that can be simultaneously mapped by the TLB.
- TLB design: 38-128 entries.
  - *Some systems small TLBs, **fully associative** (increase TLB reach by minimizing conflicting entries).
  - Other systems large TLBs, small associativity, with random/FIFO replacement policy.

[post-lecture] Hard to implement HW-level LRU. More in Ch. 5.7

Yan, Yokota

Tag, Index, and Offset

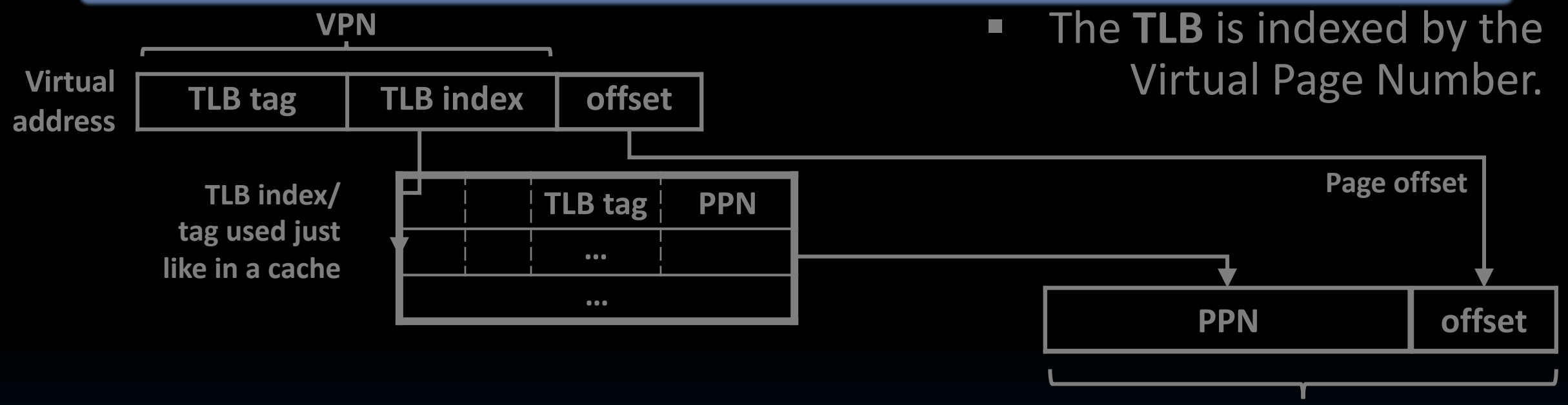
- The **TLB** is indexed by the Virtual Page Number.



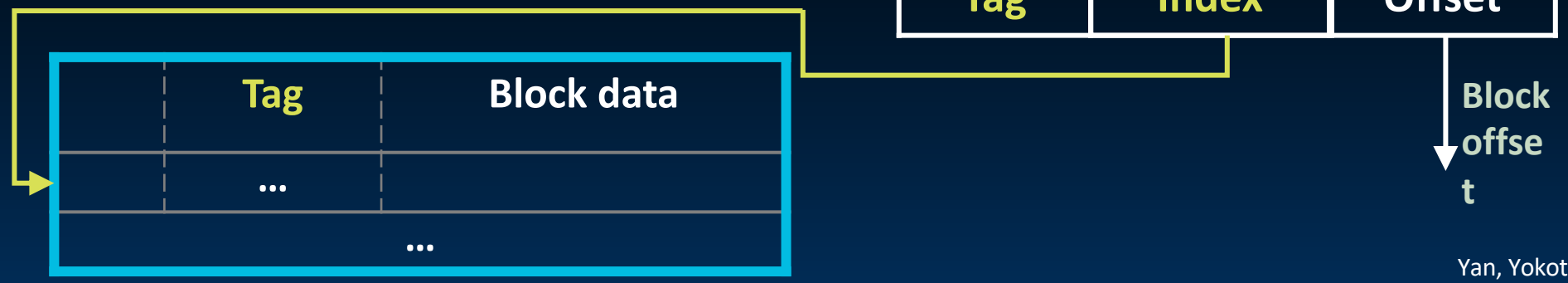
[clarification post-lecture] Assuming low associativity, e.g., direct mapped

Tag, Index, and Offset

- The **TLB** is indexed by the Virtual Page Number.



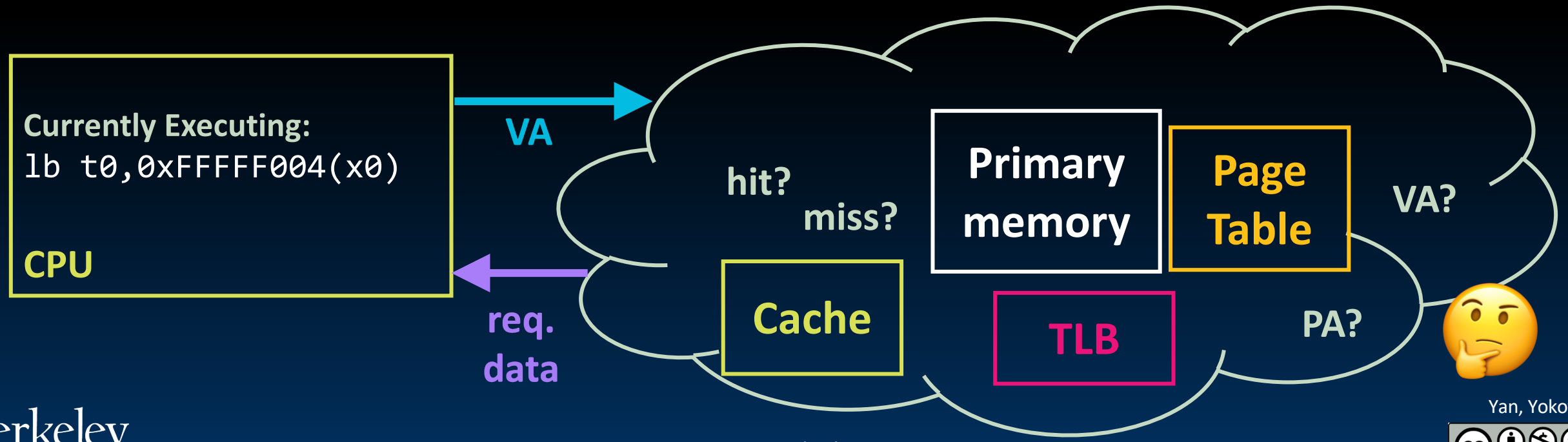
- **The data cache is indexed by the Physical Address.**



Yan, Yokota

Memory Access: TLB, Cache, DRAM, Page Table?

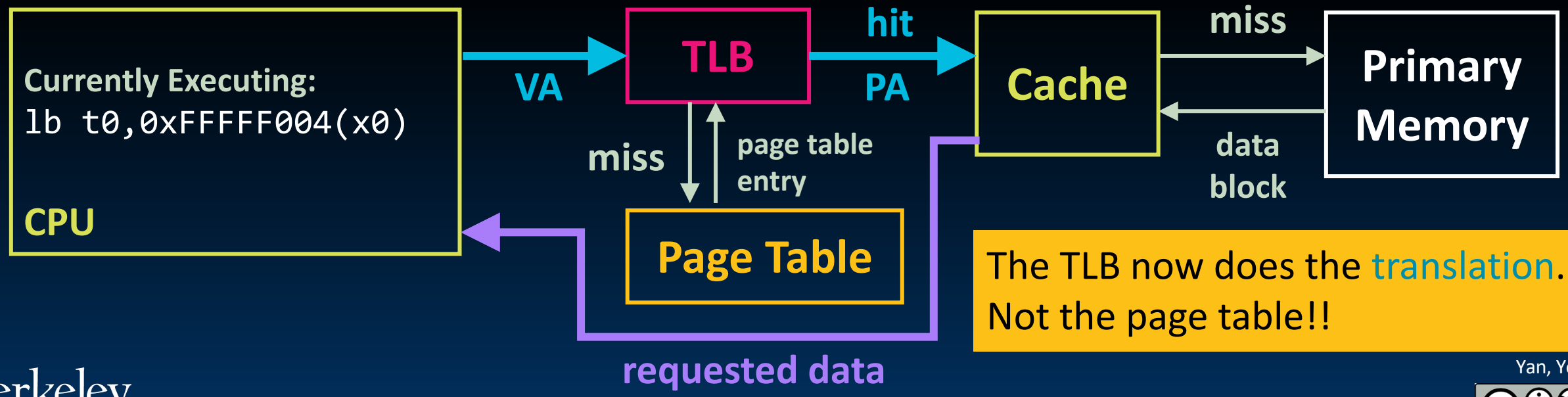
1. Can a cache hold the requested data if the corresponding page is *not* in main memory?
2. On a memory reference, which block should we access first? When should we translate virtual addresses?



Yan, Yokota

Memory Access: TLB, Cache, DRAM, Page Table

1. Can a cache hold the requested data if the corresponding page is *not* in main memory? **No!**
2. On a memory reference, which block should we access first? When should we translate virtual addresses?
   - We will assume **Physically Indexed, Physically Tagged** caches (other designs exist).
   - This means TLB first, then cache.



The TLB now does the translation. Not the page table!!

Agenda

[out of scope]

[Extra] Additional VM Details

- Page Table Details: Page Faults, Write Policy
- OS: Trap Handler
- [Review] Caches vs. Virtual Memory
- Translation Lookaside Buffer
- [Extra] Additional VM, OS Details

Yan, Yokota

System Calls and Launching Applications

- A system call (**syscall**) is a "**software interrupt**" that allows a program to request a service from the operating system.
  - Similar to a function call, except now executed by kernel.
  - Examples:
    - Creating and deleting files; reading/writing files;
    - Accessing external devices (e.g., scanner);
    - `printf`, `malloc`, etc. (`ecalls` in RISC-V); etc.
    - Launch a new process
- Suppose shell (a user process) wants to launch a new app:
  - Shell *forks* (in Linux): a syscall that traps into the OS kernel process
  - OS (supervisor mode): Load program (see CALL); jump to start of `main`. Return to user mode.
  - Shell: "wait" for `main` to return (*join*)

Yan, Yokota

Agenda

[out of scope]

Hierarchical/ Multilevel Page Tables

Yan, Yokota

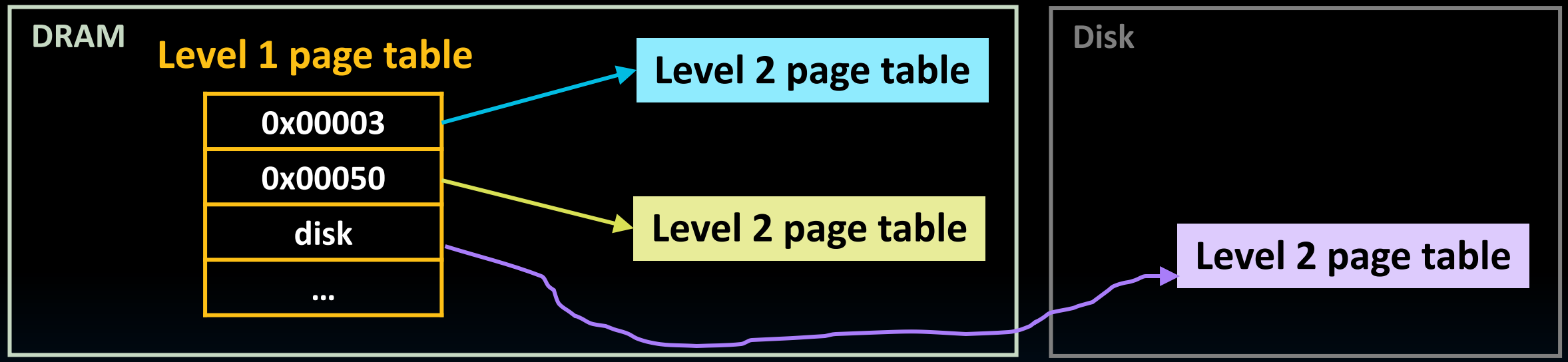
Page Tables Are Stored in Memory!

- If 32-bit virtual address space, 4 GiB RAM, 4-KiB pages:
  - # page table entries = # Virtual Page Numbers = $2^{32} / 2^{12} = 2^{20}$
  - Suppose each page table entry = 4 B (PPN + status bits).
  - Page Table Size: 4 MiB → 0.1% RAM. Not bad…
- …except *each program needs its own page table.*
- If we have 256 processes:
  - 256 x 4 MiB = $2^8 \cdot 2^2 \cdot 2^{20}$ = 1 GiB → 25% RAM just for page tables!
- Complication: **page tables must be in RAM** to be accessed.
  - Can't swap out **entire** page table to disk…

Yan, Yokota

Enter the Page Table Hierarchy

- What if we page tabled our page tables?

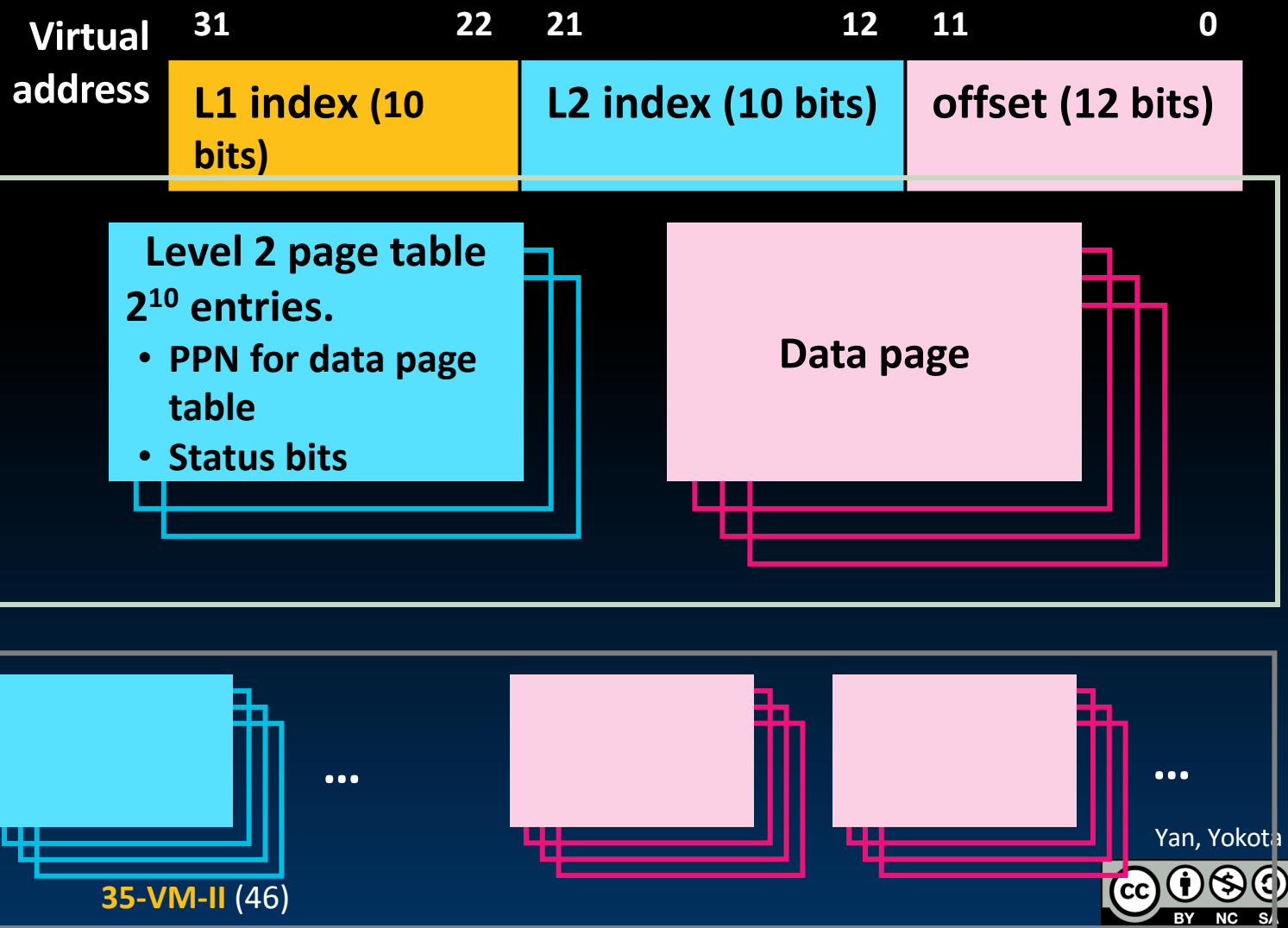


- **Multilevel page tables with decreasing page size:**
  - Key insight: **Sparsity of Virtual Address Space use.**
    - Most program use a fraction of virtual memory, so many page table entries are not accessed.
  - Level 1 page table **always** in DRAM.
  - Level 2 page tables can be in disk; loaded into DRAM via Level 1 access.

Yan, Yokota

Multilevel Page Table Translation

- 32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:
  - Page table entry size is 4 B for all levels of page tables.
  - RV32I 2-level mapping:



1-Level vs. 2-Level Page Tables

- 32-bit computer (virtual address space), 4 GiB DRAM, 4-KiB pages.
  - Page table entry size is 4 B for all levels of page tables.
  - Suppose we run 16 processes.

1. How much RAM is consumed by page tables if we have only one level of page table?

2. How much RAM is consumed by Level 1 if we use the two-level hierarchy from the previous slide?

Yan, Yokota

1-Level vs. 2-Level Page Tables

- 32-bit computer (virtual address space), 4 GiB DRAM, 4-KiB pages.
  - Page table entry size is 4 B for all levels of page tables.
  - Suppose we run 16 processes.

1. How much RAM is consumed by page tables if we have only one level of page table?

- Page offset: log(page size) = log(4KiB)
  = $\log(2^{12}) = 12$
- # entries in page table = # VPNs
  = $2^{32} / 2^{12} = 2^{20}$
- Page table size = 2^{20} x (4 B) = 2^{22} B
- Total RAM consumed =
  (16 processes) x 2^{22} B = **64 MiB**

2. How much RAM is consumed by Level 1 if we use the two-level hierarchy from the previous slide?

31 — 22	21 — 12	11 — 0
L1 index (10 bits)	**L2 index (10 bits)**	**offset (12 bits)**

- # entries in Level 1 PT (= # Level 2 PTs) = 2^{10}
- Page table size = 2^{10} x (4 B) = 2^{12} B
- Total RAM consumed =
  (16 processes) x 2^{12} B = **64 KiB**

Yan, Yokota

Multilevel Page Table Walk

- 32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:
  - RV32I 2-level mapping:

Virtual address	31 – 22	21 – 12	11 – 0
	L1 index (10 bits)	L2 index (10 bits)	offset (12 bits)

Given the page tables below,
how to translate virtual address `0x00402450` to a physical address?

Level 1 page table

0x0	disk
1	0x00020
2	0x00003
…	…

Level 2 page table

0x0	0x0FF27
1	disk
2	0x00060
…	…

Level 2 page table

0x0	0x02376
1	0x00321
2	disk
…	…

Yan, Yokota

Multilevel Page Table Walk

- 32-bit virtual address space, 4 GiB DRAM, 4-KiB pages:
  - RV32I 2-level mapping:

Virtual address	31 … 22	21 … 12	11 … 0
	L1 index (10 bits)	L2 index (10 bits)	offset (12 bits)
	0x001	0x002	0x450

Given the page tables below,
how to translate virtual address 0x00402450 to a physical address?

Level 1 page table

0x0	disk
1	0x00020
2	0x00003
…	…

Level 2 page table

0x0	0x0FF27
1	disk
2	0x00060
…	…

Level 2 page table

0x0	0x02376
1	0x00321
2	disk
…	…

Physical address	Physical Page Number	Page Offset
	0x00060	0x450

0x00060450

Page offset does not change!

Agenda

[out of scope]

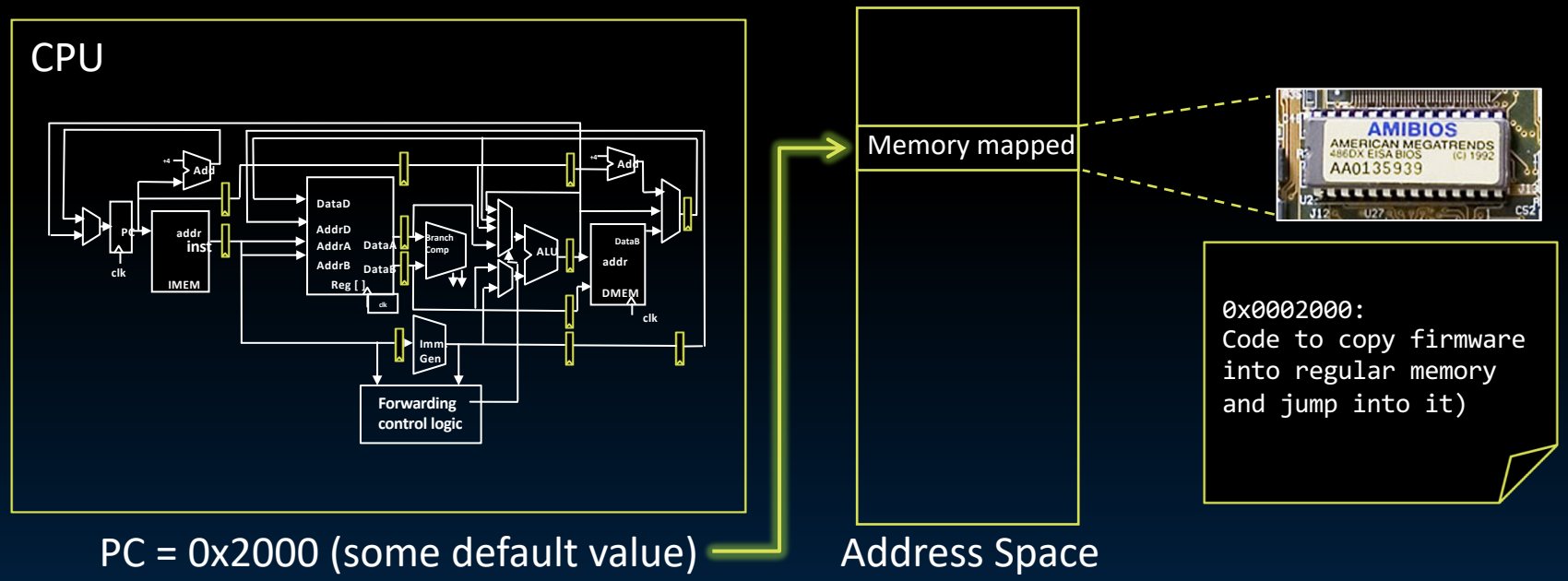
OS: Boot and System Calls

Yan, Yokota

What Happens at Boot? (1/2)

(for next time)

- When the computer switches on, it does the same as Venus:
  - The CPU executes instructions from some start address stored in Flash ROM (Read-Only Memory).



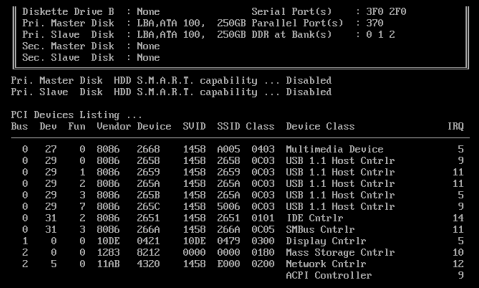
Yan, Yokota

What Happens at Boot? (2/2)

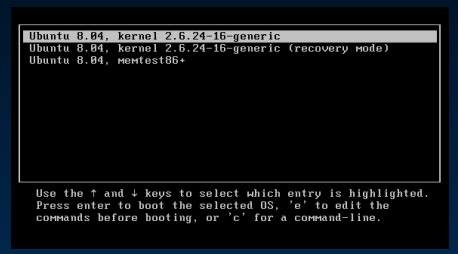
(for next time)

- Then, the **BIOS** (Basic Input Output System) firmware loads the **bootloader**, which loads the **OS kernel**.

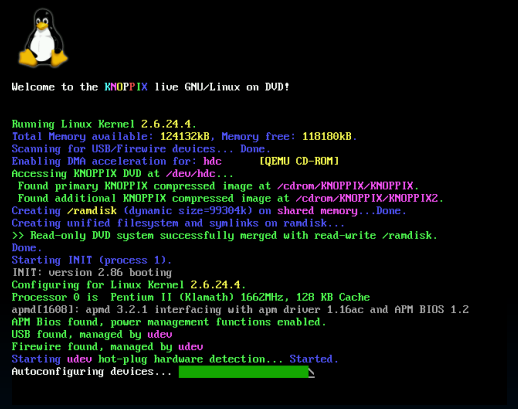
1. BIOS: Find a storage device and load the first sector (block of data).



2. Bootloader: (stored on, e.g., disk) Load the OS kernel from disk into a location in memory and jump into it.



3. OS Boot: Initialize services, drivers, etc.



4. Init: Launch an application (e.g., Terminal/Desktop/…) that waits for input in loop.

Yan, Yokota

System Calls and Launching Applications

(for next time)

- A system call (**syscall)** is a "**software interrupt**" that allows a program to request a service from the operating system.
  - Similar to a function call, except now executed by kernel.
  - Examples:
    - Creating and deleting files; reading/writing files;
    - Accessing external devices (e.g., scanner);
    - `printf`, `malloc`, etc. (`ecalls` in RISC-V); etc.
    - Launch a new process
- Suppose shell (a user process) wants to launch a new app:
  - Shell *forks* (in Linux): a *syscall* that traps into the OS kernel process
  - OS (supervisor mode): Load program (see CALL); jump to start of `main`. Return to user mode.
  - Shell: "wait" for `main` to return (*join*)

Yan, Yokota

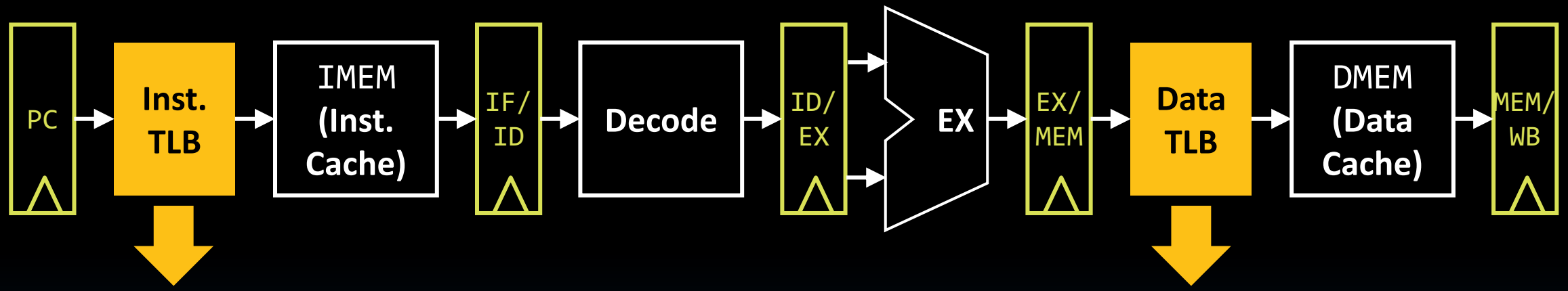
Agenda

TLBs in the Datapath

Yan, Yokota

Virtual Memory and the CPU Pipeline

Virtual Memory = address translation + protection + demand paging.



- **Each instruction/data access = address translation + functional checks.**
- **Should handle:**
  1. **TLB Miss: Needs a mechanism to refill TLB (usually done in hardware).**
  2. **Page Fault (i.e., page on disk)**

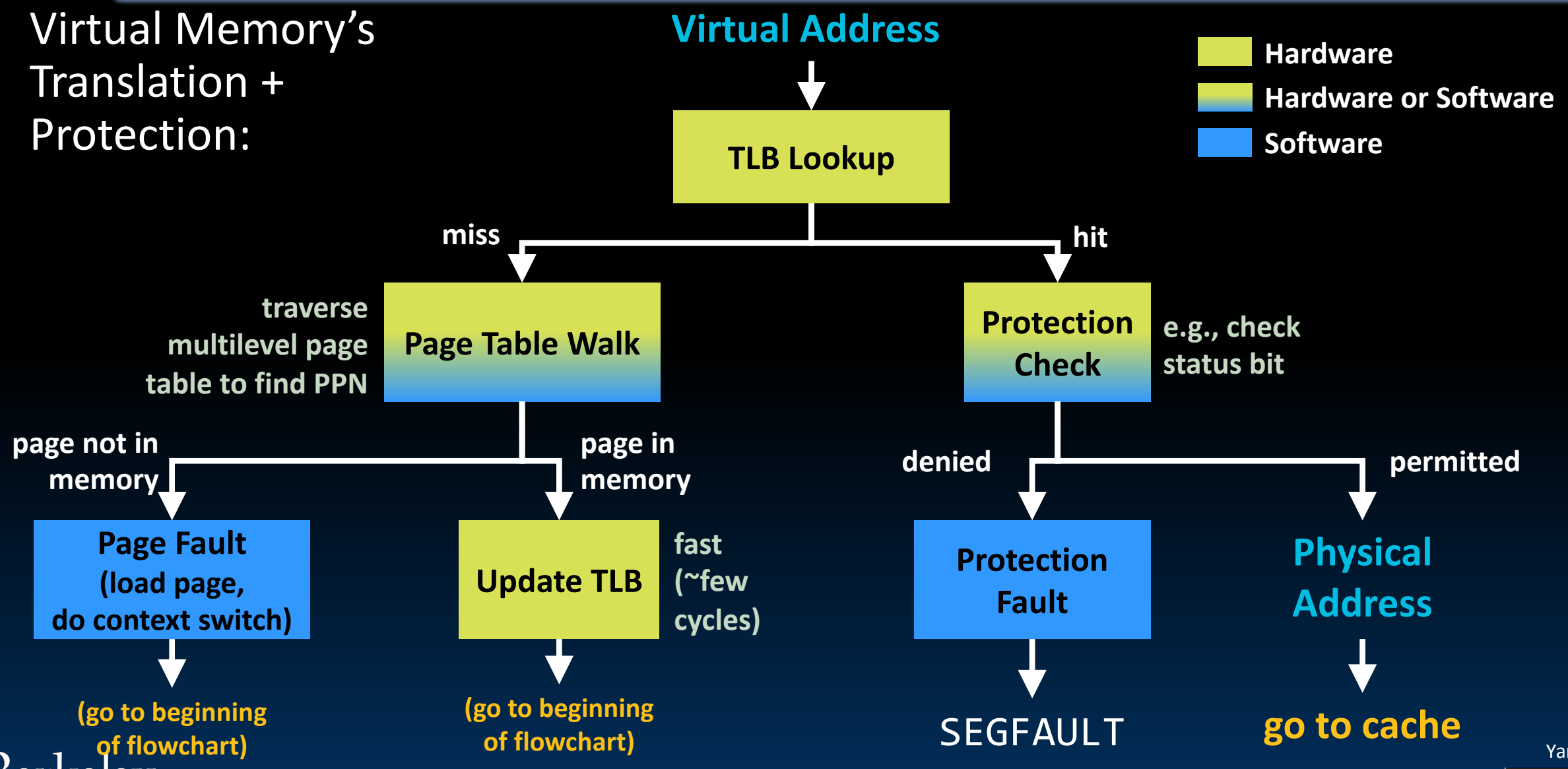
     Needs a precise trap so that software handler can easily re-execute instruction after page retrieval.
  3. **Protection violation check**

     A violation may abort the process, e.g., SEGFAULT.

Yan, Yokota

Virtual Memory Action Flowchart

Virtual Memory's Translation + Protection:



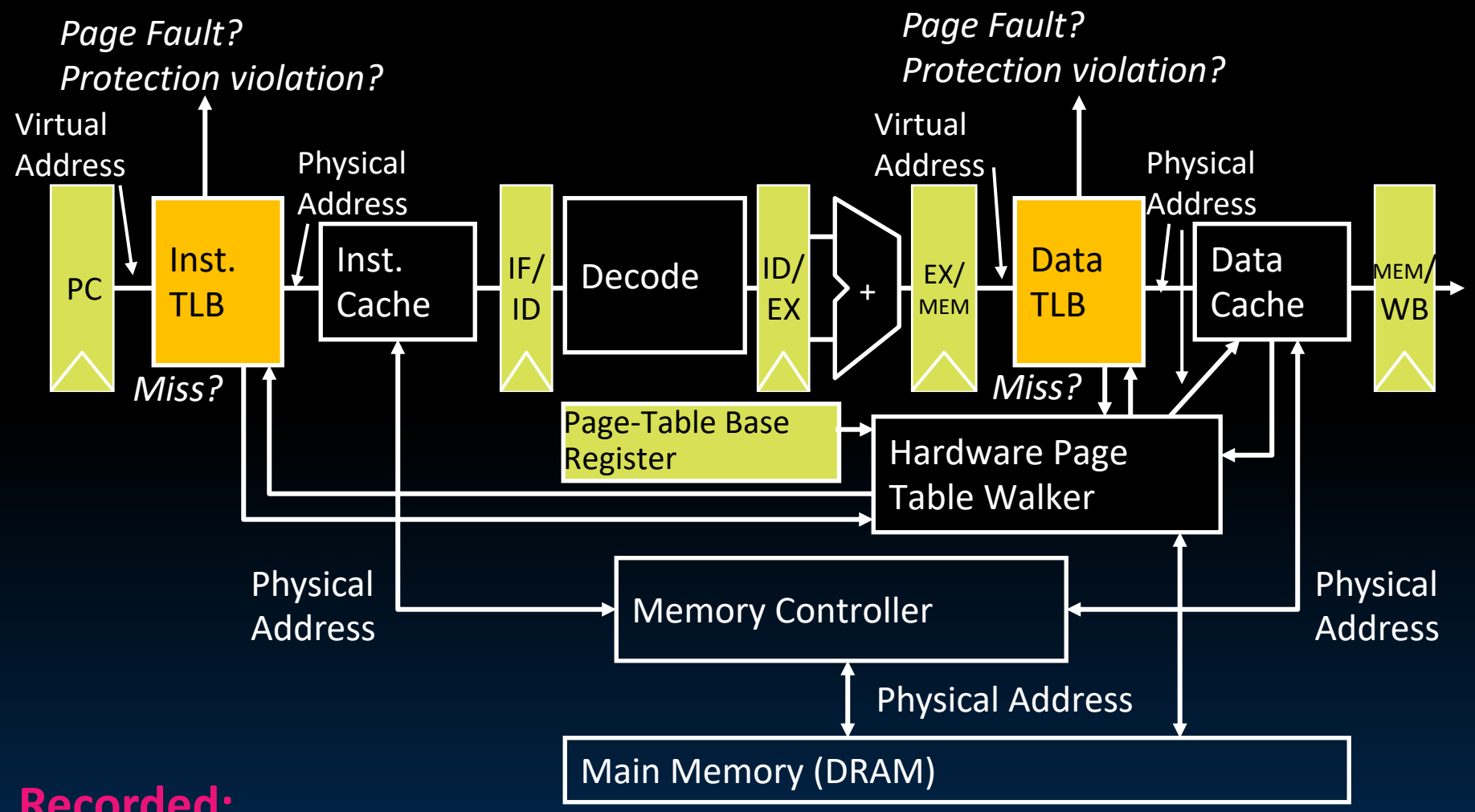
Yan, Yokota

Handling Context Switches and TLBs

- Context switches should be fast. Avoid DRAM/disk updates.
  - Keep all page tables for all currently running processes in DRAM.
  - Instead, ensure that all TLB entries refer to the *active process*.
- The high-level context switch:
  - The OS sets a timer. When it expires, perform a *hardware interrupt*.
  - Trap handler saves all register values, including:
    - Program Counter (PC)
    - *Page Table Register* (SPTBR in RV32I)
      - The memory *address* of the active process's page table.
  - Trap handler also sets all TLB entries to invalid. (other strategies exist)
  - Trap handler then loads in the next process's registers and returns to user mode.

Yan, Yokota

A Full, Page-Based Virtual Memory Machine (optional)



(Assume page tables are held in untranslated physical memory)

Recorded:
https://youtu.be/eVIsejli9hU

Yan, Yokota

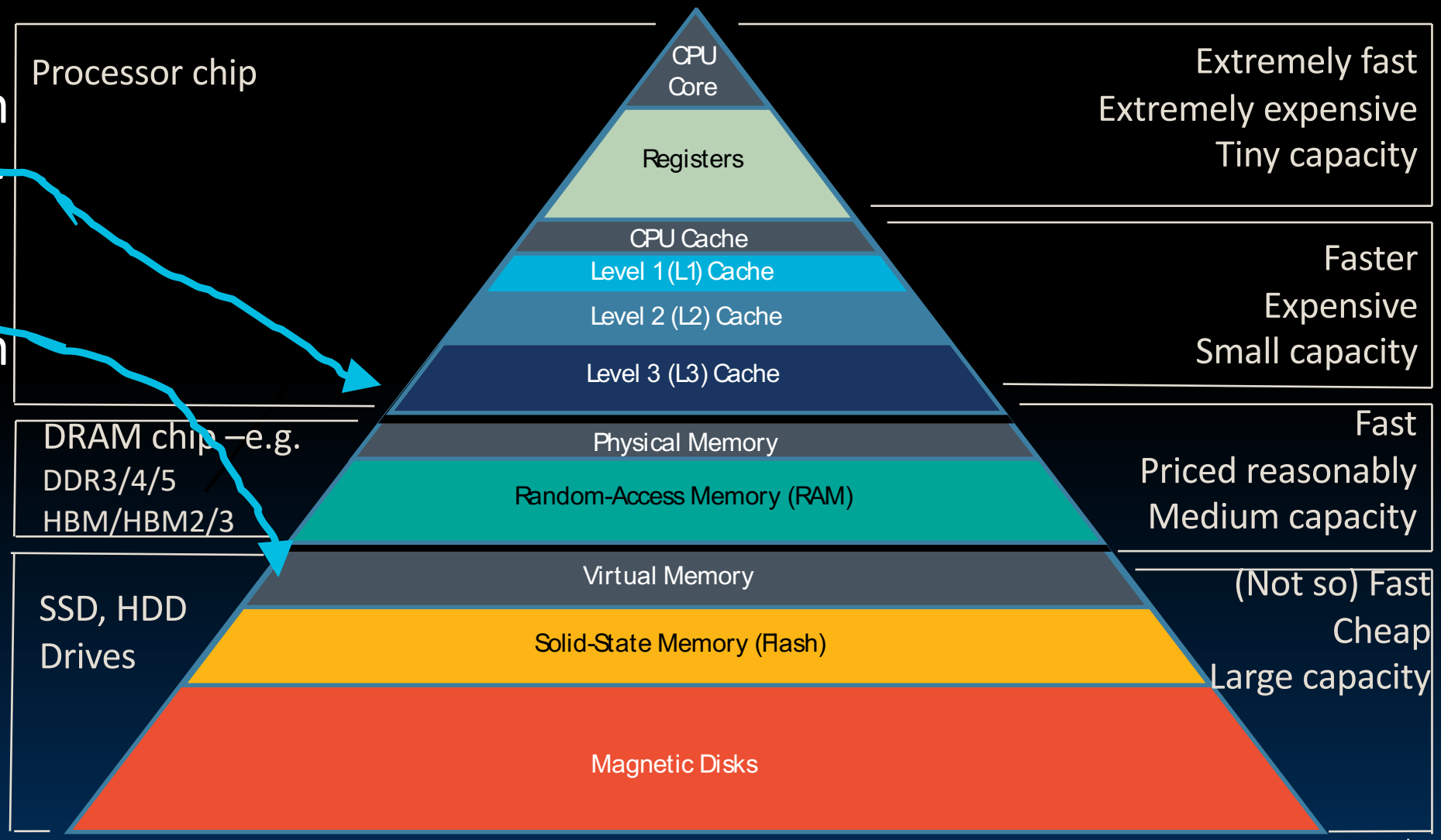
Agenda

Virtual Memory Performance

Yan, Yokota

The Entire Modern Memory Hierarchy

- Cache policies manages memory between cache and DRAM.
- Virtual Memory manages memory between DRAM and disk.
  - TLB comes *before* the cache but affects transfer of data between DRAM/disk.



Yan, Yokota

Average Memory Access Time (AMAT)

- Recall: Performance is **Average Memory Access Time (AMAT)**.

$$AMAT = (hit\ time) + (miss\ rate) \times (miss\ penalty)$$

- Previously, we treated main memory as the lowest level.
- Now with demand paging (virtual memory):
  - Disk is lowest level.
  - Main memory is like a mid-level cache.
- Main memory also has a hit rate:
  - Hit rate = 1 – Page Fault Rate

🤔 Design question: What is a reasonable hit rate?

Yan, Yokota

AMAT, DRAM only

L1 cache

Hit Time: 1 cycle

Hit Rate: 95%

L2 cache

Hit Time: 10 cycles

Hit Rate: 60%
(of L1 misses)

DRAM

Hit Time: 200 cycles
(≈ 100 ns if 2GHz)

- **Average Memory Access Time with DRAM only:**

$$AMAT_{no} = 1 + 5\% \times (\text{L1 miss penalty})$$
$$= 1 + 5\% \times \left(10 + 40\% \times \underbrace{(\text{L2 miss penalty})}_{\text{200 cycles}}\right) = \boxed{5.5 \text{ clock cycles}}$$

Yan, Yokota

AMAT w/Demand Paging

L1 cache
Hit Time: 1 cycle
Hit Rate: 95%

L2 cache
Hit Time: 10 cycles
Hit Rate: 60%
(of L1 misses)

DRAM
Hit Time: 200 cycles
(≈ 100 ns if 2GHz)
Hit Rate: HR_{mem}???

Disk
Hit Time:
20,000,000 cycles
(≈ 10 ms if 2GHz)

- Average Memory Access Time with DRAM only:

$$AMAT_{no} = 1 + 5\% \times (\text{L1 miss penalty})$$
$$= 1 + 5\% \times \left(10 + 40\% \times \underbrace{(\text{L2 miss penalty})}_{\text{200 cycles}}\right) = \boxed{5.5 \text{ clock cycles}}$$

- **Average Memory Access Time with demand paging:**

$$AMAT_{dp} = 1 + 5\% \times \left(10 + 40\% \times (200 + (1 - HR_{mem}) \times 20{,}000{,}000)\right)$$
$$= \underbrace{\boxed{5.5}}_{AMAT_{no}} + \underbrace{(5\% \times 40\% \times (1 - HR_{mem}) \times 20{,}000{,}000)}_{\text{performance cost of demand paging (disk access)}} \quad \text{(distributive property)}$$

Yan, Yokota

AMAT w/Demand Paging

L1 cache	L2 cache	DRAM	Disk
Hit Time: 1 cycle	Hit Time: 10 cycles	Hit Time: 200 cycles (≈ 100 ns if 2GHz)	Hit Time: 20,000,000 cycles (≈ 10 ms if 2GHz)
Hit Rate: 95%	Hit Rate: 60% (of L1 misses)	Hit Rate: HR_{mem}???	

- **Average Memory Access Time with demand paging:**

$$AMAT_{dp} = 5.5 + (5\% \times 40\% \times (1 - HR_{mem}) \times 20{,}000{,}000)$$

A. $HR_{mem} = 99\%$

B. $HR_{mem} = 99.9\%$

C. $HR_{mem} = 99.9999\%$

Which proposed DRAM hit rate minimizes the performance cost of disk?

Yan, Yokota

AMAT w/Demand Paging

L1 cache	L2 cache	DRAM	Disk
Hit Time: 1 cycle	Hit Time: 10 cycles	Hit Time: 200 cycles (≈ 100 ns if 2GHz)	Hit Time: 20,000,000 cycles (≈ 10 ms if 2GHz)
Hit Rate: 95%	Hit Rate: 60% (of L1 misses)	Hit Rate: HR_{mem}???	

- **Average Memory Access Time with demand paging:**

$$AMAT_{dp} = 5.5 + (5\% \times 40\% \times (1 - HR_{mem}) \times 20{,}000{,}000)$$

A. $HR_{mem} = 99\%$ $\quad AMAT_A = 5.5 + (0.02 \times (.01) \times 20{,}000{,}000) = 4{,}005.5$ **cycles**

1 in 20,000 memory accesses goes to disk → 10 second program takes 20 hours!!

B. $HR_{mem} = 99.9\%$ $\quad AMAT_B = 5.5 + (0.02 \times (.001) \times 20{,}000{,}000) = 405.5$ **cycles**

C. $HR_{mem} = 99.9999\%$ $\quad AMAT_C = 5.5 + (0.02 \times (.000001) \times 20{,}000{,}000)$
$= 5.9$ **cycles** ✅

A reasonable page fault rate is ≪ 0.01%.