Virtual Memory III
CSE 351 Autumn 2021

Instructor:

Justin Hsia

Teaching Assistants:

Allie Pfleger
Anirudh Kumar
Assaf Vayner
Atharva Deodhar
Celeste Zeng
Dominick Ta
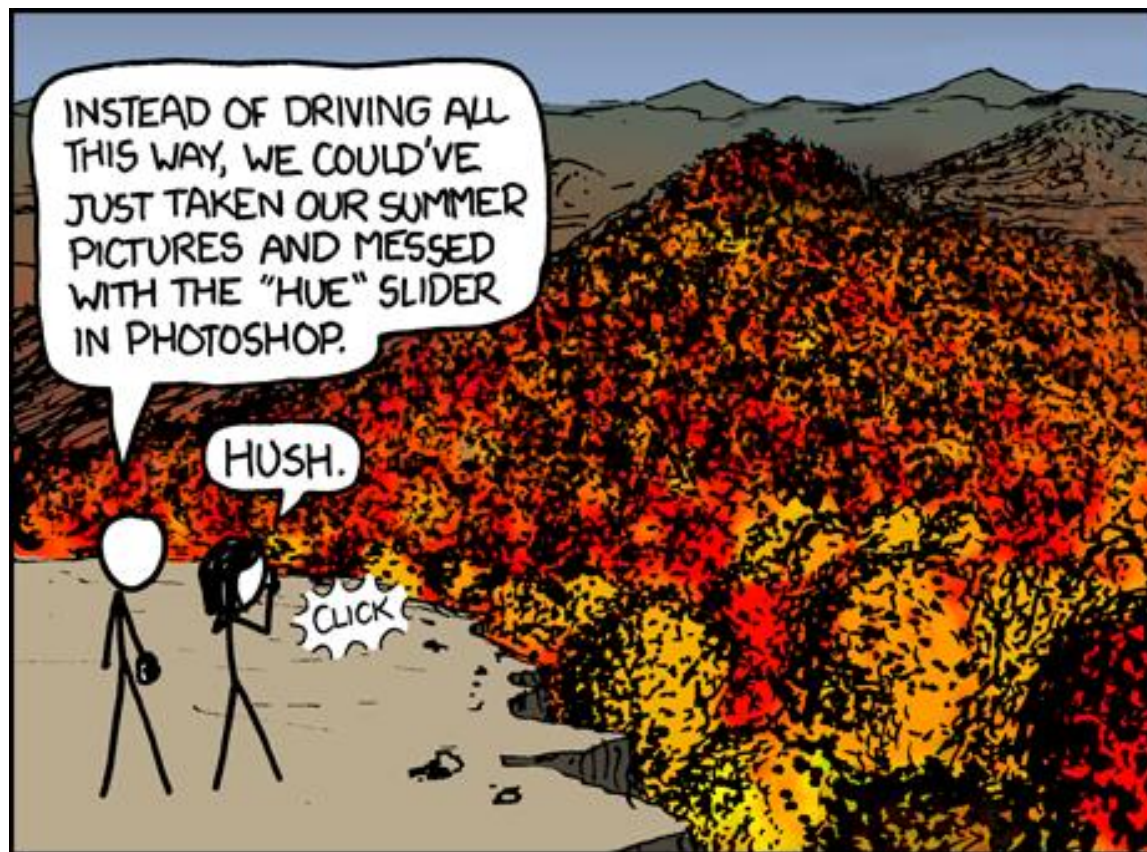Francesca Wang
Hamsa Shankar
Isabella Nguyen
Joy Dang
Julia Wang
Maggie Jiang
Monty Nitschke
Morel Fotsing
Sanjana Chintalapati

https://xkcd.com/648/

Relevant Course Information

❖ hw21 due Friday (11/26)

❖ Lab 4 due Monday (11/29)

❖ hw22 due Wednesday (12/1)

❖ "Virtual Section" on Virtual Memory

▪ Worksheet and solutions released on Wednesday or Thursday

▪ Videos will be released of material review and problem solutions
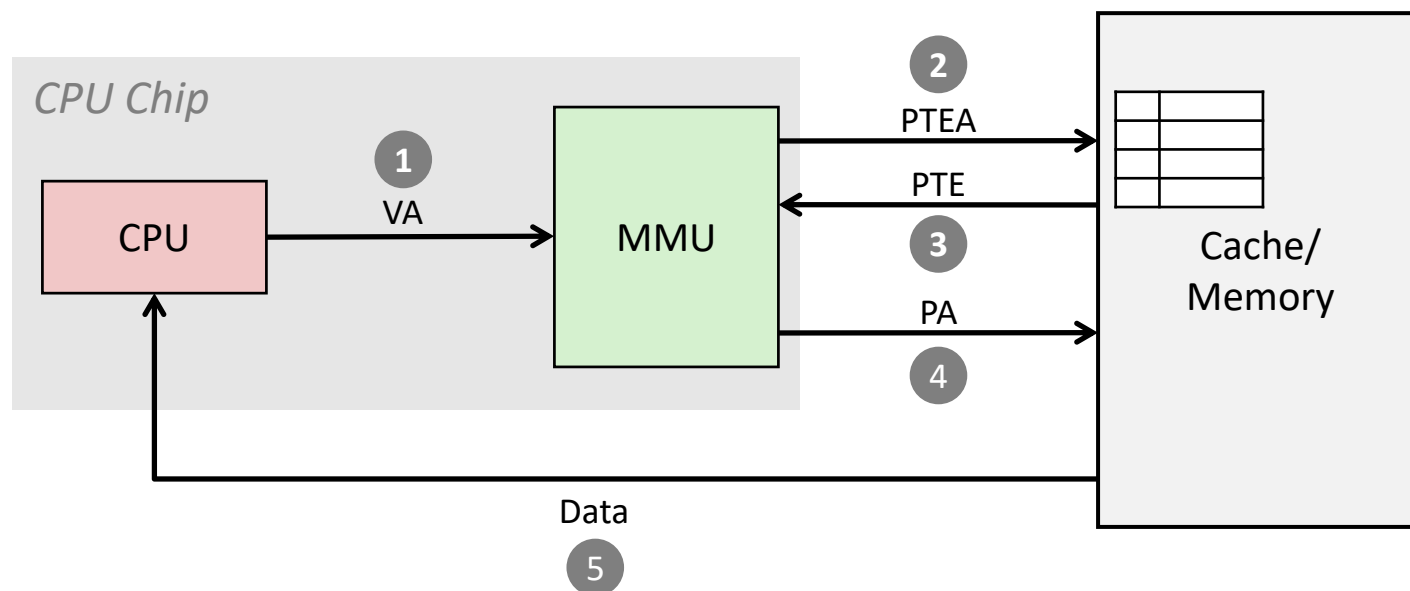
❖ Final Dec. 13-15, regrade requests Dec. 18-19

Reading Review

❖ Terminology:

  ▪ Address translation: page hit, page fault

  ▪ Translation Lookaside Buffer (TLB): TLB Hit, TLB Miss

❖ Questions from the Reading?

Address Translation: Page Hit

1) Processor sends *virtual* address to MMU (*memory management unit*)

2-3) MMU fetches PTE from page table in cache/memory
    (Uses PTBR to find beginning of page table for current process)

4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data to processor

VA = Virtual Address   PTEA = Page Table Entry Address   PTE= Page Table Entry
PA = Physical Address   Data = Contents of memory stored at VA originally requested by CPU

4

Address Translation: Page Fault
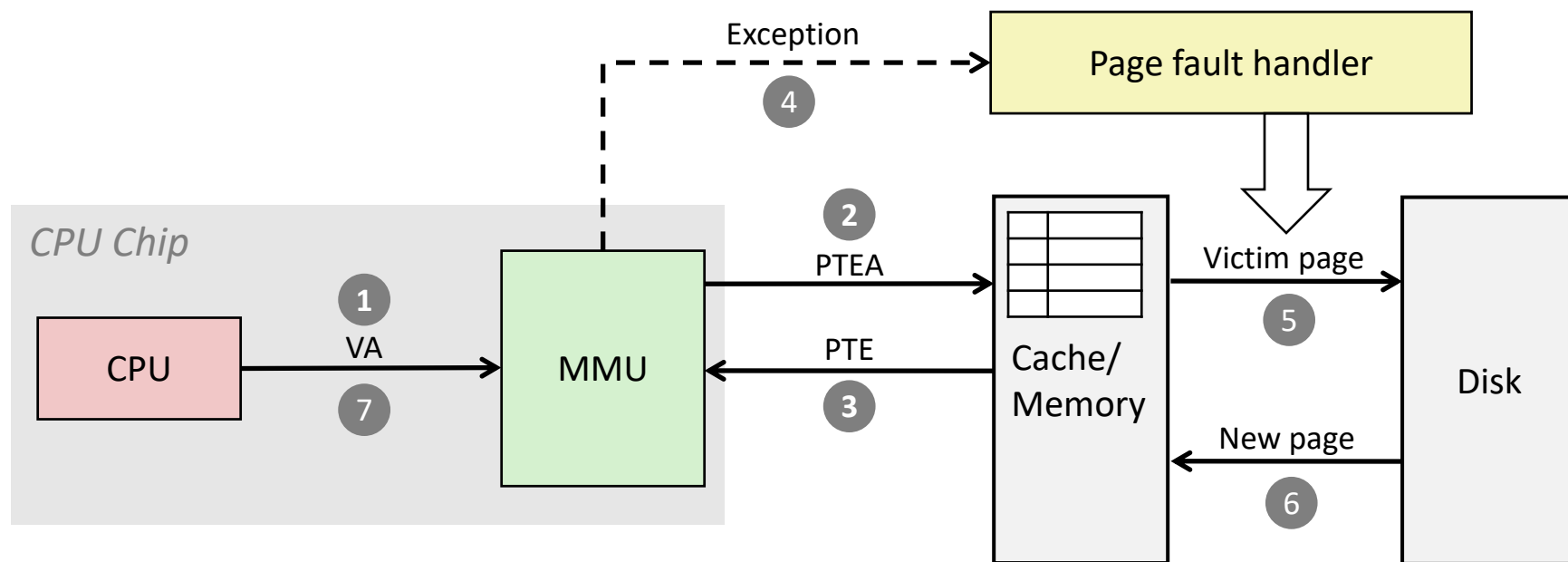
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in cache/memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

Hmm… Translation Sounds Slow

❖ The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request

- The PTEs *may* be cached in L1 like any other memory word
  - But they may be evicted by other data references
  - And a hit in the L1 cache still requires 1-3 cycles

❖ *What can we do to make this faster?*

- **Solution:** add another cache! 🎉

Speeding up Translation with a TLB

❖ *Translation Lookaside Buffer* (TLB):

- Small hardware cache in MMU
  - Split VPN into **TLB Tag** and **TLB Index** based on # of sets in TLB
- Maps virtual page numbers to physical page numbers
- Stores *page table entries* for a small number of pages
  - Modern Intel processors have 128 or 256 entries in TLB
- Much faster than a page table lookup in cache/memory

Virtual Page Number	Page offset

TLBT	TLBI

TLB

Set			
0	V	TLBT	PTE
	V	TLBT	PTE
1	V	TLBT	PTE
	V	TLBT	PTE

TLB Hit

TLB

VPN	→	PTE
VPN	→	PTE
VPN	→	PTE

CPU Chip

TLB

2 VPN

PTE **3**

1 VA

CPU → MMU → PA **4** → Cache/ Memory

Data **5**
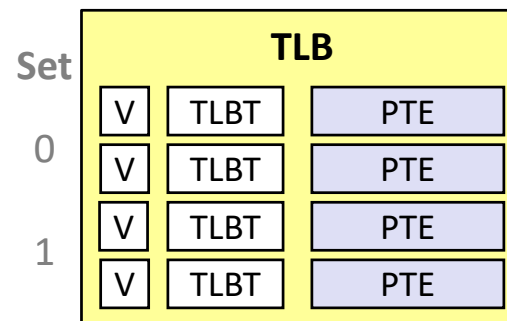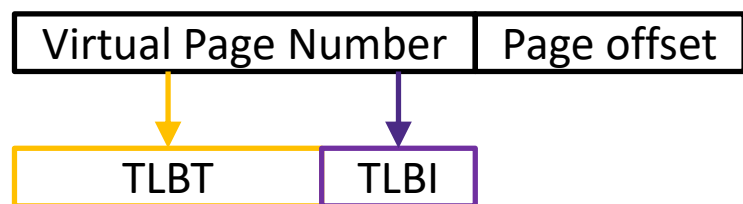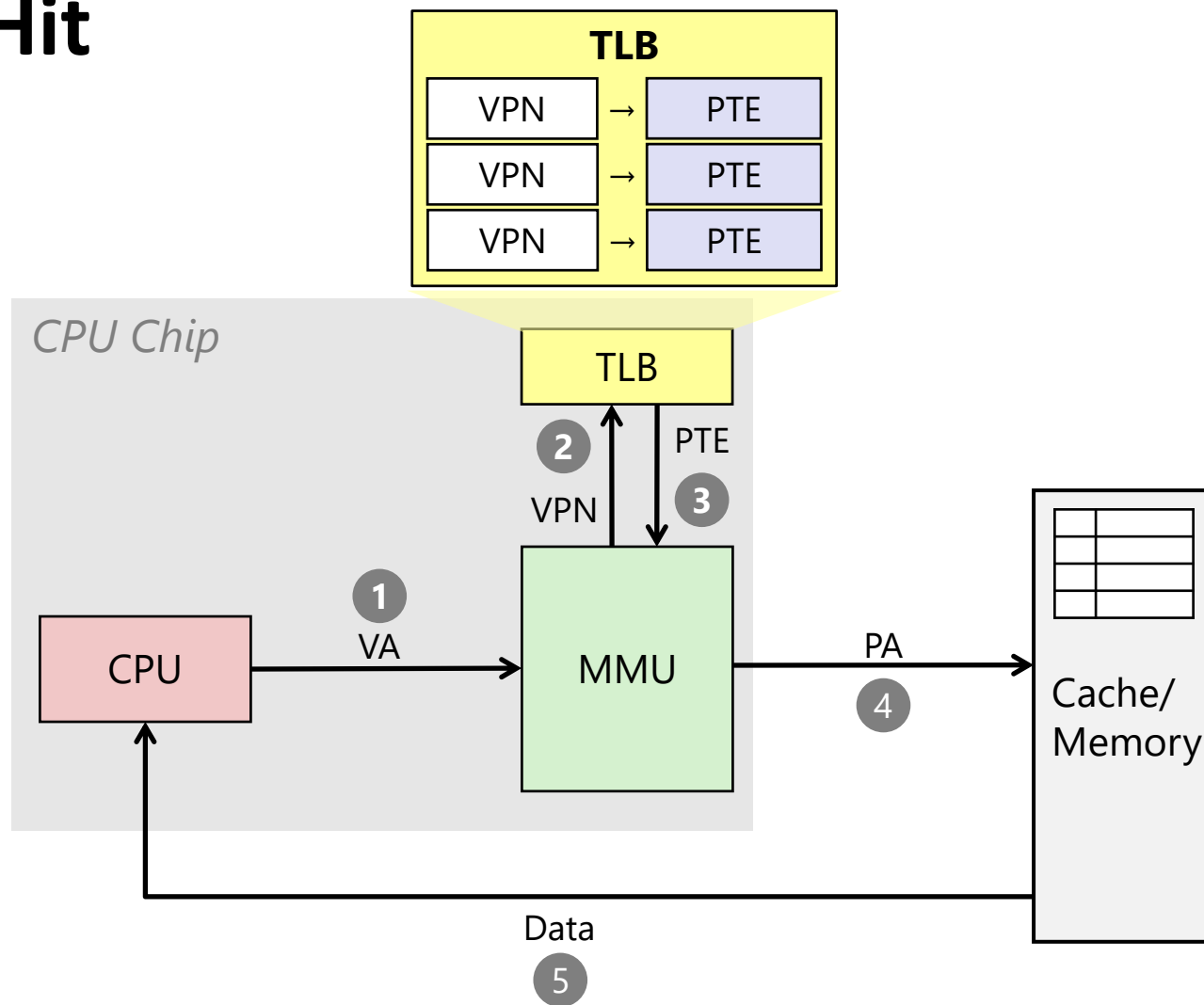
❖ A TLB hit eliminates a memory access!

TLB Miss
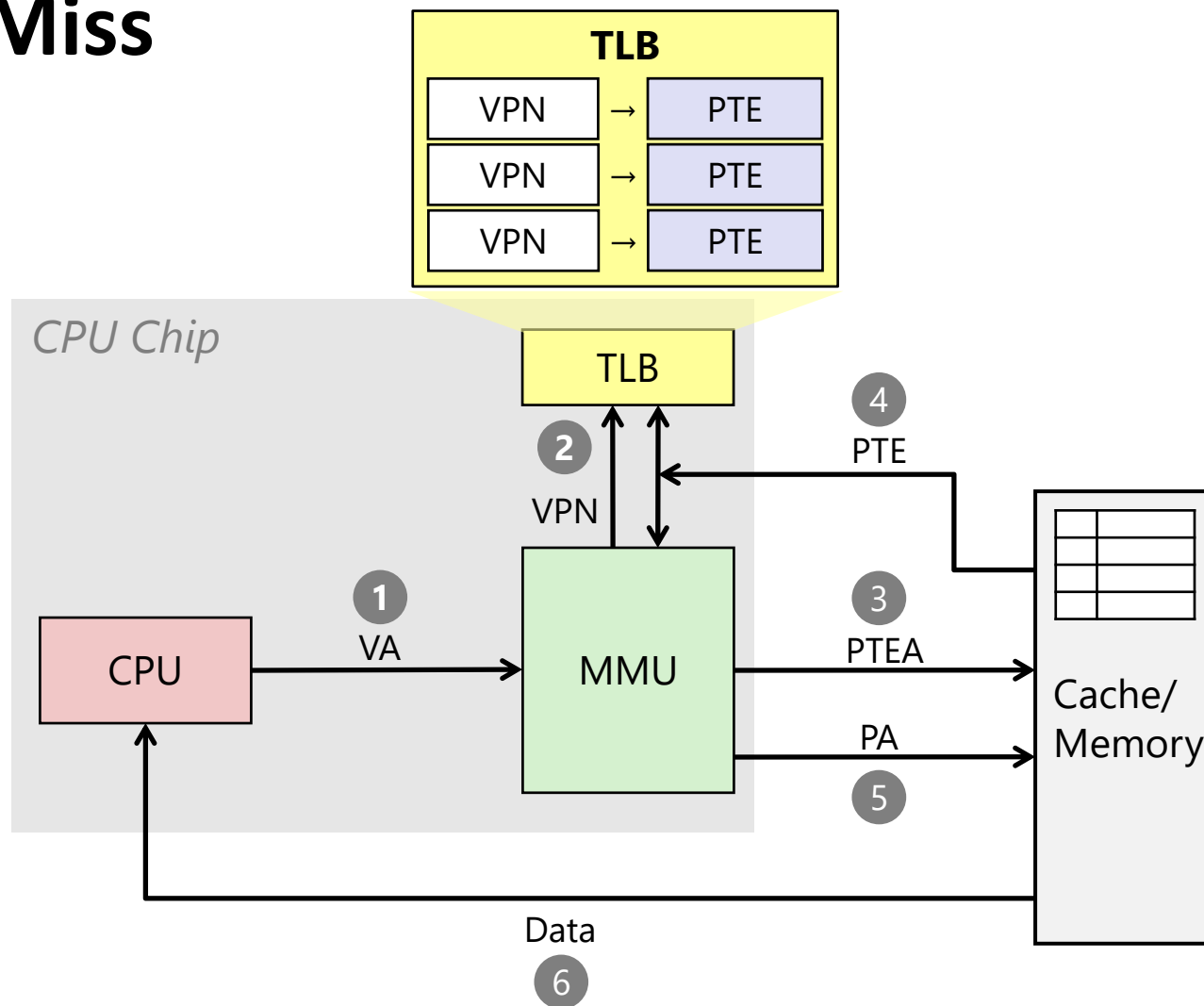
- ❖ A TLB miss incurs an additional memory access (the PTE)
  - ▪ Fortunately, TLB misses are rare

Fetching Data on a Memory Read

1) Check TLB

   ▪ <u>Input</u>: VPN, <u>Output</u>: PPN

   ▪ *TLB Hit:* Fetch translation, return PPN

   ▪ *TLB Miss:* Check page table (in memory)

      • *Page Table Hit:* Load page table entry into TLB

      • *Page Fault:* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

2) Check cache

   ▪ <u>Input</u>: physical address, <u>Output</u>: data

   ▪ *Cache Hit:* Return data value to processor

   ▪ *Cache Miss:* Fetch data value from memory, store it in cache, return it to processor

Address Translation

Address Manipulation

request from CPU: n-bit virtual address

split to access TLB: | TLB Tag | TLB Index | Page Offset |

(on TLB miss) access PT: | Virtual Page Number | Page offset |

TRANSLATION

m-bit physical address: | Physical Page Number | Page offset |

split to access cache: | Cache Tag | Cache Index | Block offset |
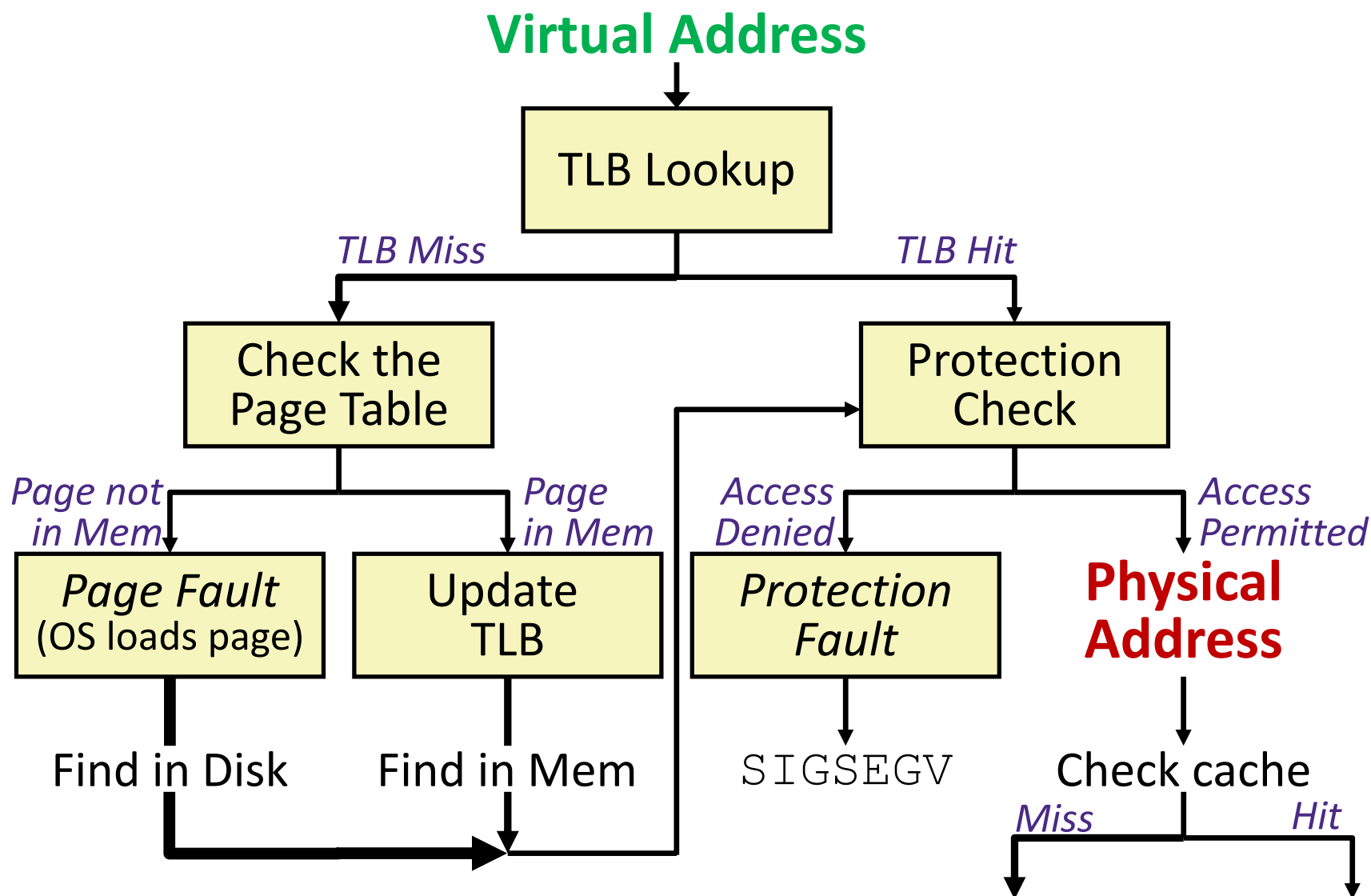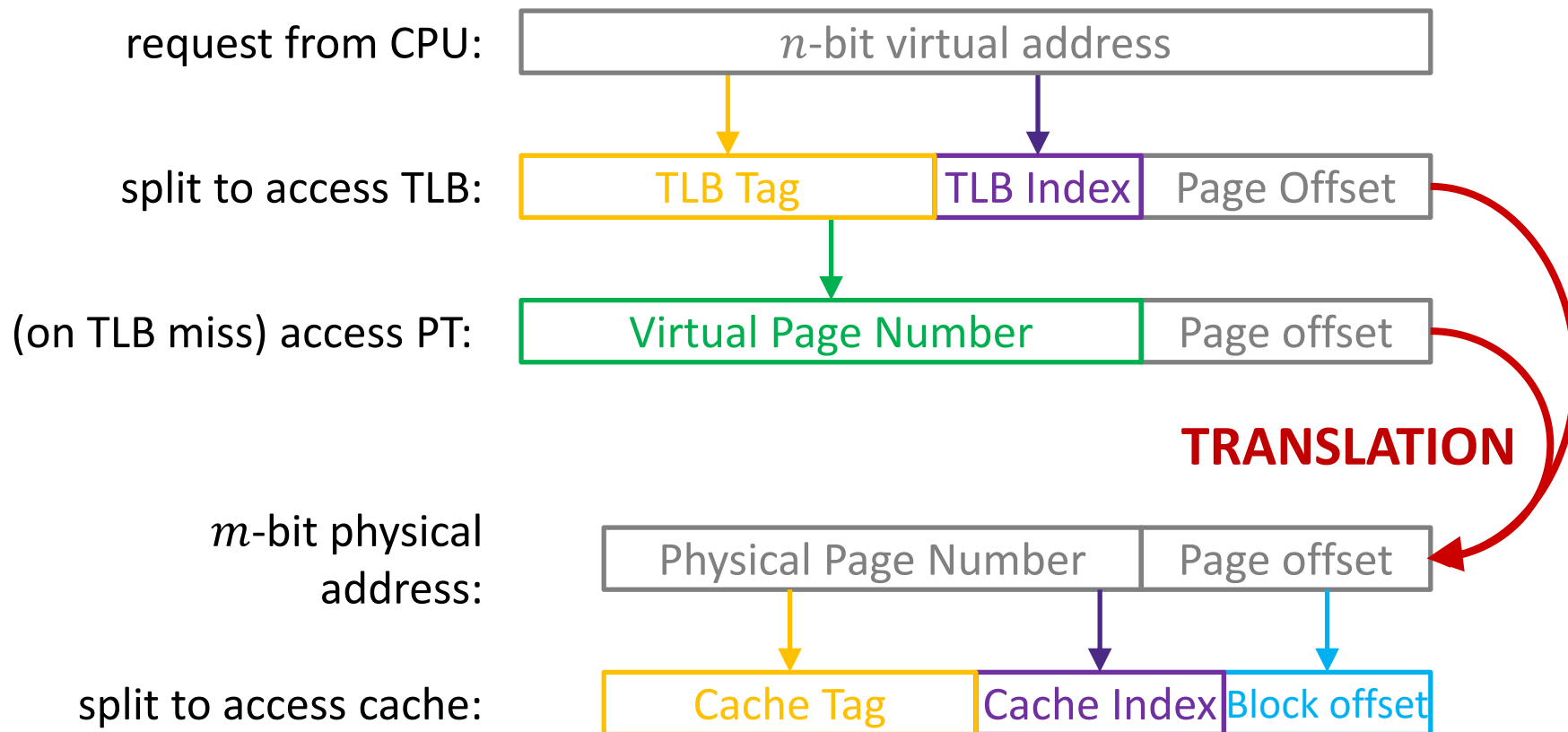
Context Switching Revisited

❖ What needs to happen when the CPU switches processes?

- Registers:
  - Save state of old process, load state of new process
  - Including the Page Table Base Register (PTBR)

- Memory:
  - Nothing to do! Pages for processes already exist in memory/disk and protected from each other

- TLB:
  - *Invalidate* all entries in TLB – mapping is for old process' VAs

- Cache:
  - Can leave alone because storing based on PAs – good for shared data

Summary of Address Translation Symbols

❖ **Basic Parameters**
  ▪ $N = 2^n$ Number of addresses in virtual address space
  ▪ $M = 2^m$ Number of addresses in physical address space
  ▪ $P\ = 2^p$ Page size (bytes)

❖ **Components of the virtual address (VA)**
  ▪ **VPO**   Virtual page offset
  ▪ **VPN**   Virtual page number
  ▪ **TLBI**   TLB index
  ▪ **TLBT**   TLB tag

❖ **Components of the physical address (PA)**
  ▪ **PPO**   Physical page offset (same as VPO)
  ▪ **PPN**   Physical page number

Simple Memory System Example (small)

❖ Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes

13	12	11	10	9	8	7	6	5	4	3	2	1	0

← **VPN** → ← **VPO** →

Virtual Page Number   Virtual Page Offset

11	10	9	8	7	6	5	4	3	2	1	0

← **PPN** → ← **PPO** →

Physical Page Number   Physical Page Offset

15
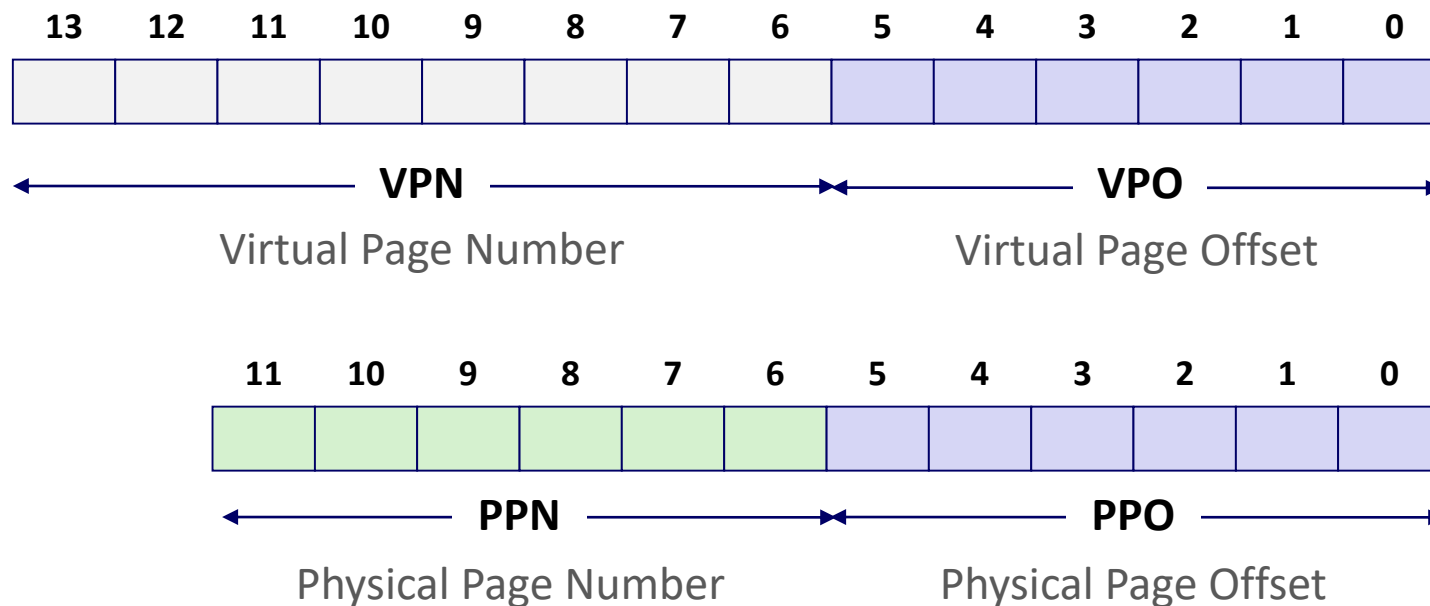
Simple Memory System: Page Table

❖ Only showing first 16 entries (out of _____)
- **Note:** showing 2 hex digits for PPN even though only 6 bits
- **Note:** other management bits not shown, but part of PTE

VPN	PPN	Valid
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

❖ 16 entries total

❖ 4-way set associative

Why does the TLB ignore the page offset?

TLB tag	→	TLB index

13	12	11	10	9	8	7	6	5	4	3	2	1	0

← virtual page number → ← virtual page offset →
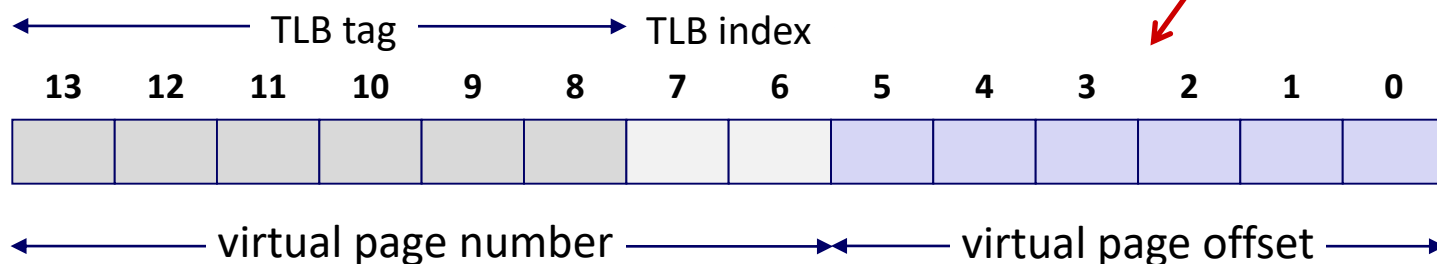
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: Cache

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Direct-mapped with K = 4 B, C/K = 16

❖ Physically addressed
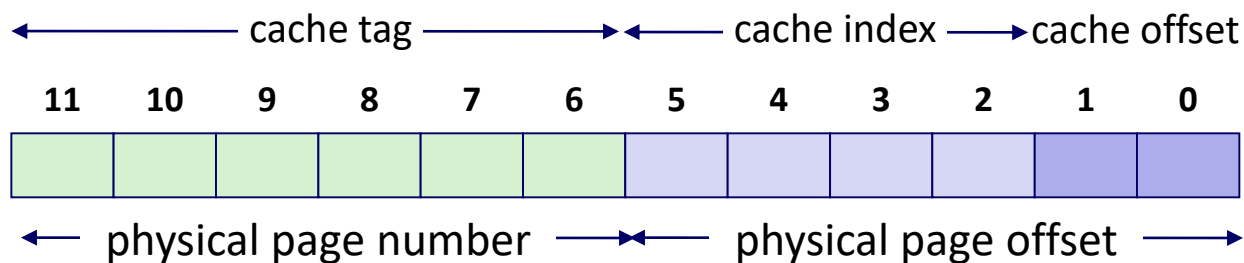
Index	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Current State of Memory System

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Page table (partial):

VPN	PPN	V	VPN	PPN	V
0	28	1	8	13	1
1	–	0	9	17	1
2	33	1	A	09	1
3	02	1	B	–	0
4	–	0	C	–	0
5	16	1	D	2D	1
6	–	0	E	–	0
7	–	0	F	0D	1

Cache:

Index	Tag	V	B0	B1	B2	B3	Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	–	–	–	–

Memory Request Example #1

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: `0x03D4`
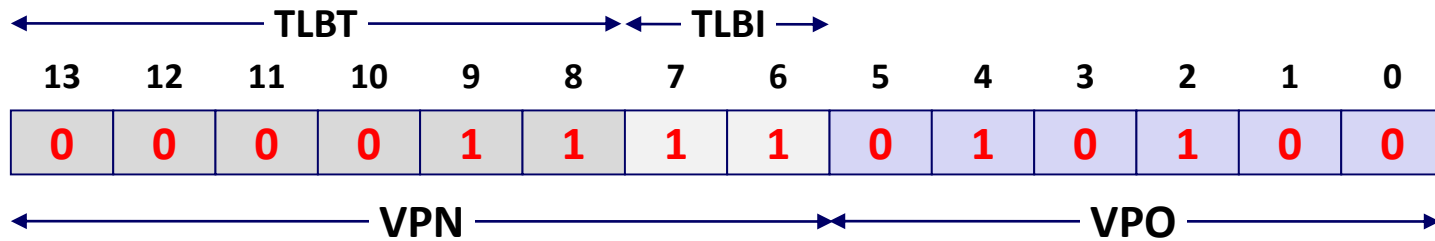
	TLBT						TLBI							
13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN							VPO						

VPN _____   TLBT _____   TLBI _____   TLB Hit? ___   Page Fault? ___   PPN _____

❖ Physical Address:

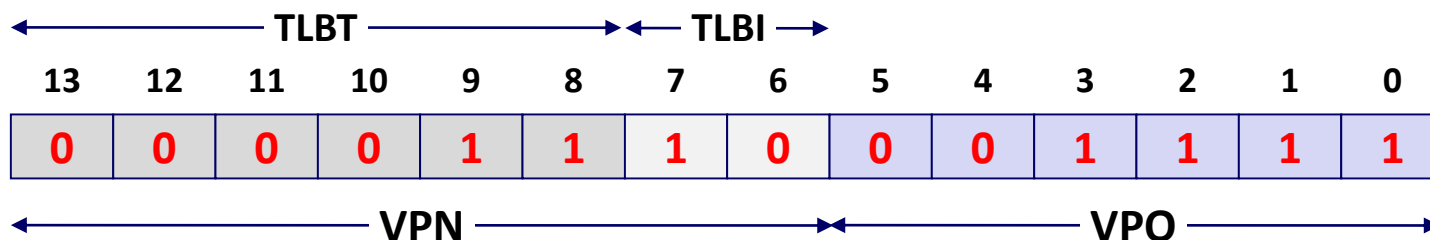|	CT						CI				CO	
11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	PPN						PPO					
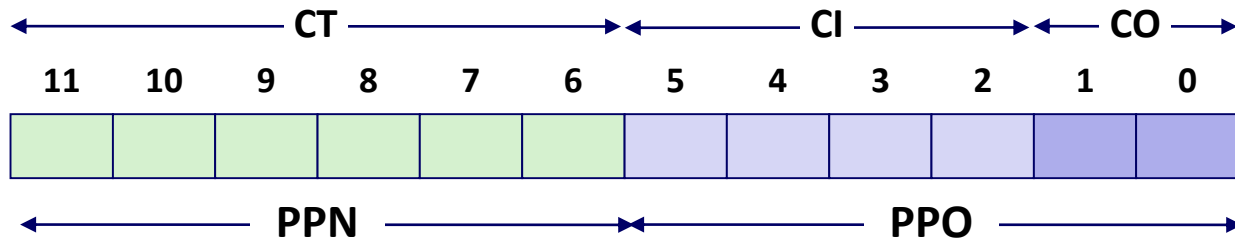
CT _____   CI _____   CO _____   Cache Hit? ___   Data (byte) _____

Memory Request Example #2

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: `0x038F`

	TLBT →					← TLBI →							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	0	1	1	1	1

← VPN → | ← VPO →

VPN _____   TLBT _____   TLBI _____   TLB Hit? ____   Page Fault? ____   PPN _____

❖ Physical Address:

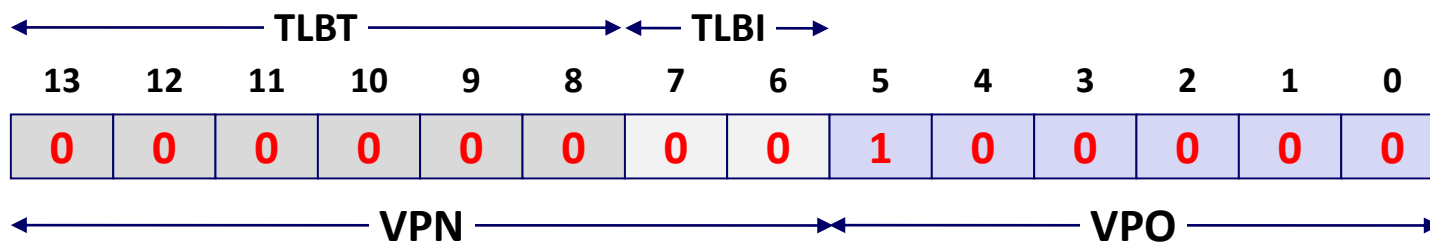|	CT →					← CI →				← CO →	
11	10	9	8	7	6	5	4	3	2	1	0

← PPN → | ← PPO →
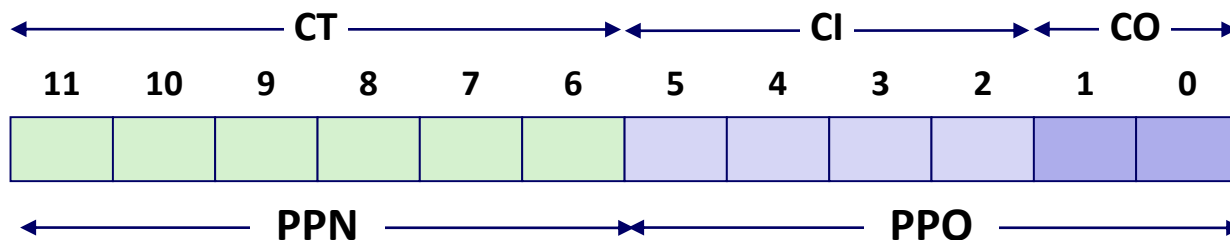
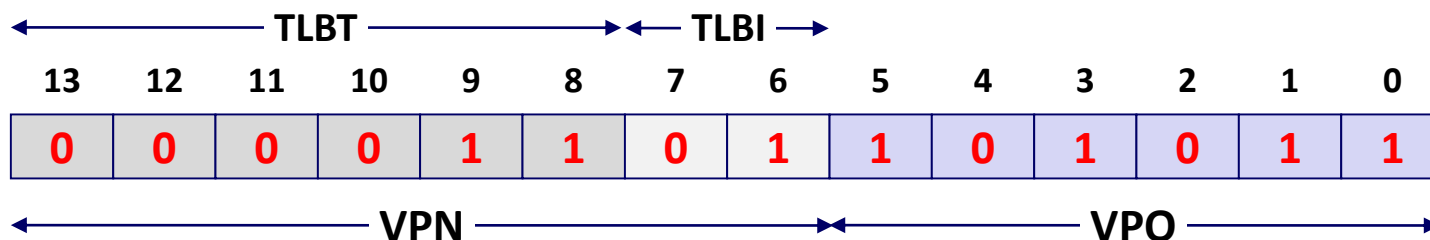CT _____   CI _____   CO _____   Cache Hit? ____   Data (byte) _____

Memory Request Example #3

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: `0x0020`

	TLBT					TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0
			VPN							VPO			

VPN _____   TLBT _____   TLBI _____   TLB Hit? ___   Page Fault? ___   PPN _____

❖ Physical Address:

	CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
		PPN						PPO			
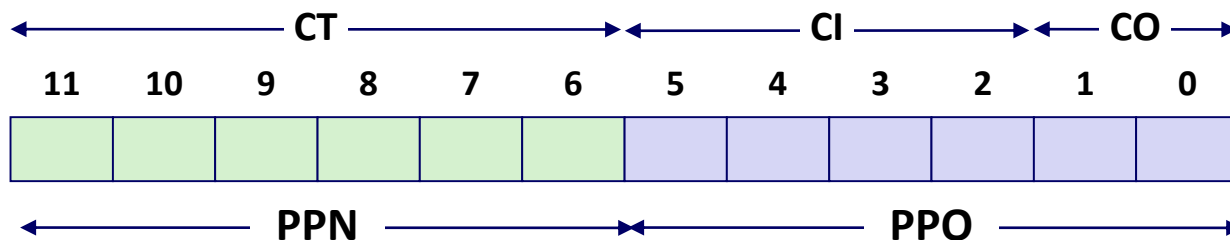
CT _____   CI _____   CO _____   Cache Hit? ___   Data (byte) _____

Memory Request Example #4

Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: `0x036B`

	TLBT →					← TLBI →							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1

← VPN → ← VPO →

VPN _____   TLBT _____   TLBI _____   TLB Hit? ___   Page Fault? ___   PPN _____

❖ Physical Address:

	CT →					← CI →				← CO →	
11	10	9	8	7	6	5	4	3	2	1	0

← PPN → ← PPO →

CT _____   CI _____   CO _____   Cache Hit? ___   Data (byte) _____

UNIVERSITY *of* WASHINGTON
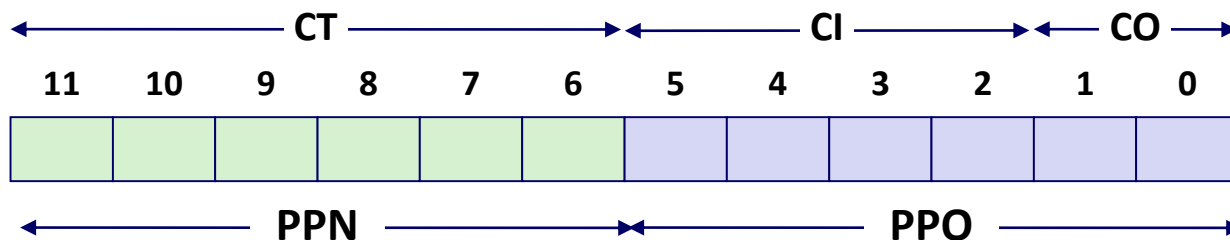
Memory Overview

❖ `movl 0x8043ab, %rdi`

Disk

Page

requested 32-bits

CPU

MMU

TLB

Cache

Line

Main memory (DRAM)

Page

Block

24
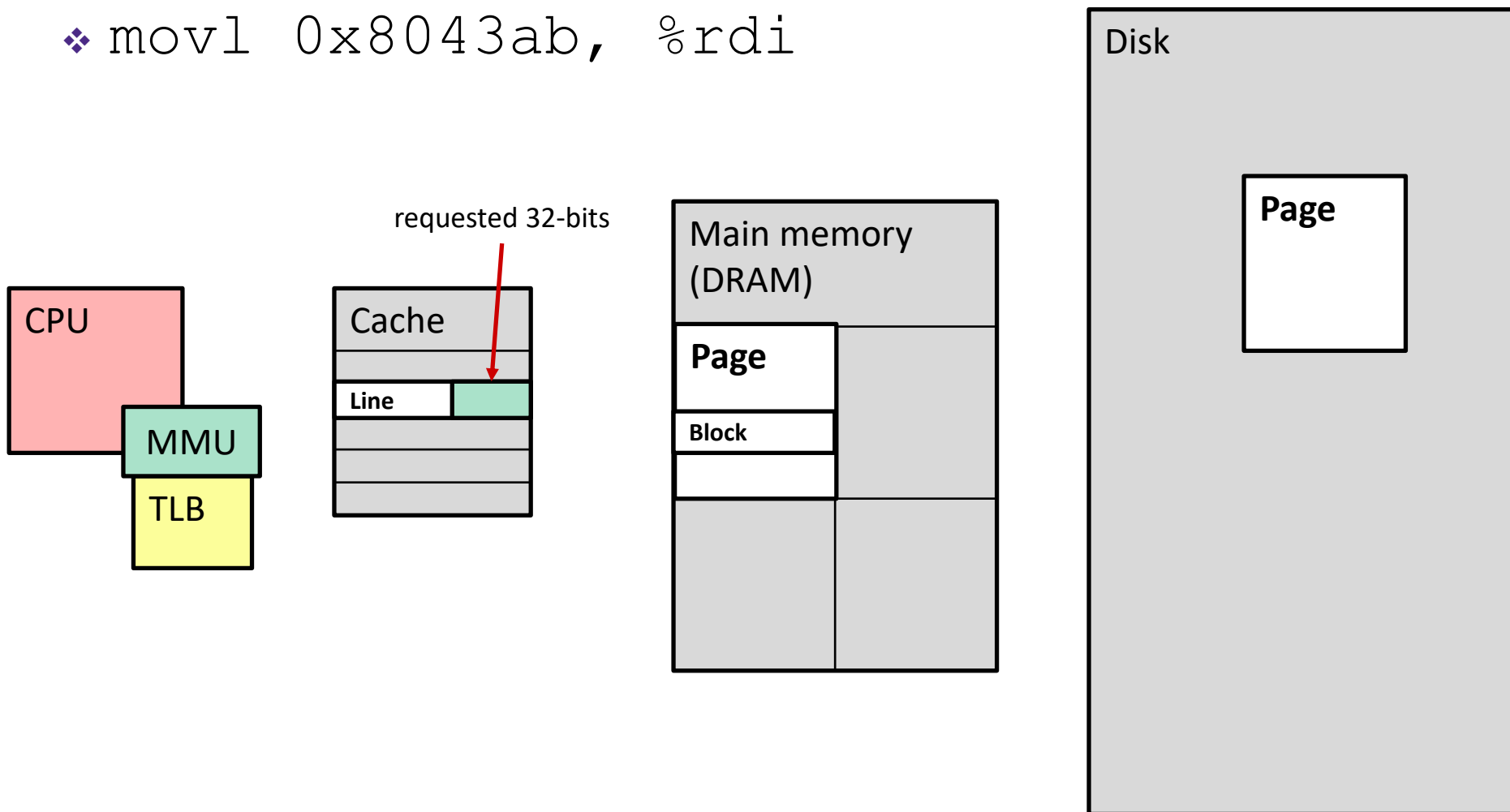
Virtual Memory Summary

❖ Programmer's view of virtual memory

- Each process has its own private linear address space
- Cannot be corrupted by other processes

❖ System view of virtual memory

- Uses memory efficiently by caching virtual memory pages
  - Efficient only because of locality
- Simplifies memory management and sharing
- Simplifies protection by providing permissions checking

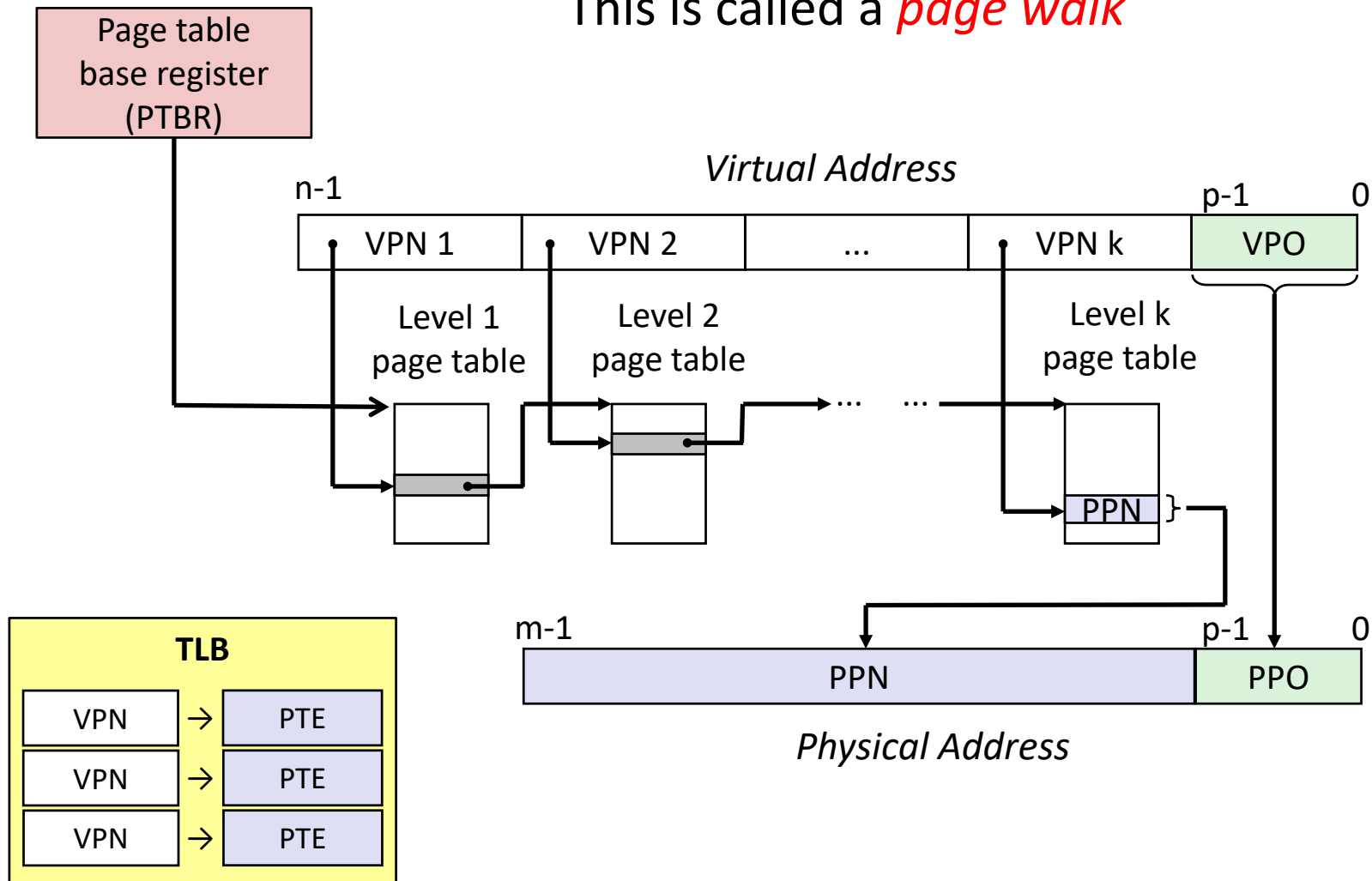# BONUS SLIDES

❖ Multi-level Page Tables

Page Table Reality

This is extra (non-testable) material

❖ Just one issue... the numbers don't work out for the story so far!

❖ The problem is the page table for each process:
- Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
- How many page table entries is that?

- About how long is each PTE?

- **Moral:** Cannot use this naïve implementation of the virtual→physical page mapping – it's *way* too big

A Solution: Multi-level Page Tables

This is extra (non-testable) material

This is called a *page walk*

Page table base register (PTBR)

Virtual Address

n-1			p-1	0
VPN 1	VPN 2	...	VPN k	VPO

Level 1 page table
Level 2 page table
Level k page table

PPN

TLB

VPN	→	PTE
VPN	→	PTE
VPN	→	PTE

m-1		p-1	0
	PPN		PPO

Physical Address

28

Multi-level Page Tables

- ❖ A tree of depth k where each node at depth i has up to 2^j children if part i of the VPN has j bits

- ❖ Hardware for multi-level page tables inherently more complicated
  - But it's a necessary complexity – 1-level does not fit

- ❖ Why it works: Most subtrees are not used at all, so they are never created and definitely aren't in physical memory
  - Parts created can be evicted from cache/memory when not being used
  - Each node can have a size of ~1-100KB

- ❖ But now for a k-level page table, a TLB miss requires $k + 1$ cache/memory accesses
 - Fine so long as TLB misses are rare – motivates larger TLBs

BONUS SLIDES

For Fun: DRAMMER Security Attack

❖ Why are we talking about this?

- **Recent:** First announced in October 2016; latest attack variant announced November 2021

- **Relevant:** Uses your system's memory setup to gain elevated privileges

  - Ties together some of what we've learned about virtual memory and processes

- **Interesting:** It's a software attack that uses *only hardware vulnerabilities* and requires *no user permissions*

Underlying Vulnerability: Row Hammer

❖ Dynamic RAM (DRAM) has gotten denser over time

  ▪ DRAM cells physically closer and use smaller charges

  ▪ More susceptible to "*disturbance errors*" (interference)

❖ DRAM capacitors need to be "refreshed" periodically (~64 ms)

  ▪ Lose data when loss of power

  ▪ Capacitors accessed in rows

❖ Rapid accesses to one row can flip bits in an adjacent row!

  ▪ ~ 100K to 1M times

DRAM cells
Activation target rows
Victim row
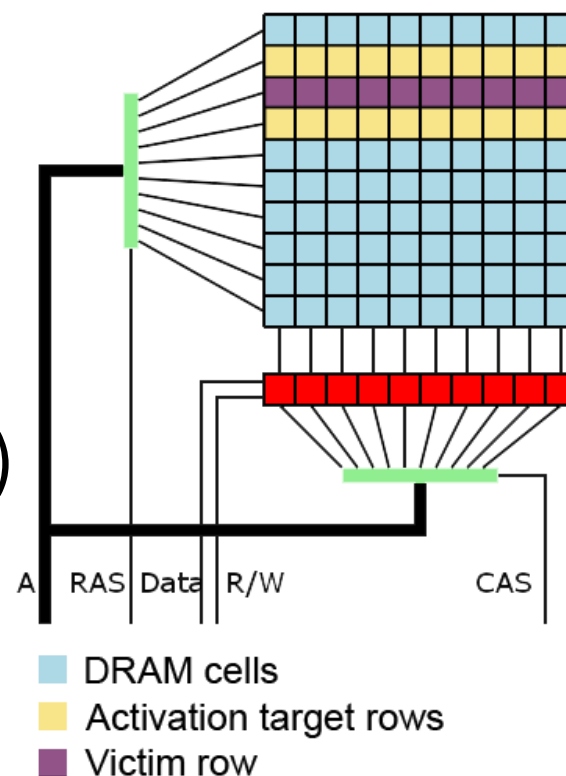
By Dsimic (modified), CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=38868341

31

Row Hammer Exploit

❖ **Force constant memory access**

- ▪ Read then flush the cache
- ▪ `clflush` – flush cache line
  - • Invalidates cache line containing the specified address
  - • Not available in all machines or environments

```
hammertime:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  jmp hammertime
```

- ▪ Want addresses `X` and `Y` to fall in activation target row(s)
  - • Good to understand how *banks* of DRAM cells are laid out

❖ **The row hammer effect was discovered in 2014**

- ▪ Only works on certain types of DRAM (2010 onwards)
- ▪ These techniques target x86 machines

Consequences of Row Hammer

❖ Row hammering process can affect another process via memory

- Circumvents virtual memory protection scheme
- Memory needs to be in an adjacent row of DRAM

❖ Worse: privilege escalation

- Page tables live in memory!
- Hope to change PPN to access other parts of memory, or change permission bits
- **Goal:** gain read/write access to a page containing a page table, hence granting process read/write access to *all of physical memory*

Effectiveness?

❖ **Doesn't seem so bad – random bit flip in a row of physical memory**

 ▪ Vulnerability affected by system setup and physical condition of memory cells

❖ **Improvements:**

  ▪ Double-sided row hammering increases speed & chance

  ▪ Do system identification first (*e.g.*, Lab 4)

   • Use timing to infer memory row layout & find "bad" rows

   • Allocate a huge chunk of memory and try many addresses, looking for a reliable/repeatable bit flip

  ▪ Fill up memory with page tables first

   • `fork` extra processes; hope to elevate privileges in any page table

What's DRAMMER?

❖ No one previously made a huge fuss

- **Prevention:** error-correcting codes, target row refresh, higher DRAM refresh rates
- Often relied on special memory management features
- Often crashed system instead of gaining control

❖ Research group found a *deterministic* way to induce row hammer exploit in a non-x86 system (ARM)

- Relies on predictable reuse patterns of standard physical memory allocators
- Universiteit Amsterdam, Graz University of Technology, and University of California, Santa Barbara

DRAMMER Demo Video

❖ It's a shell, so not that sexy-looking, but still interesting
  ▪ Apologies that the text is so small on the video
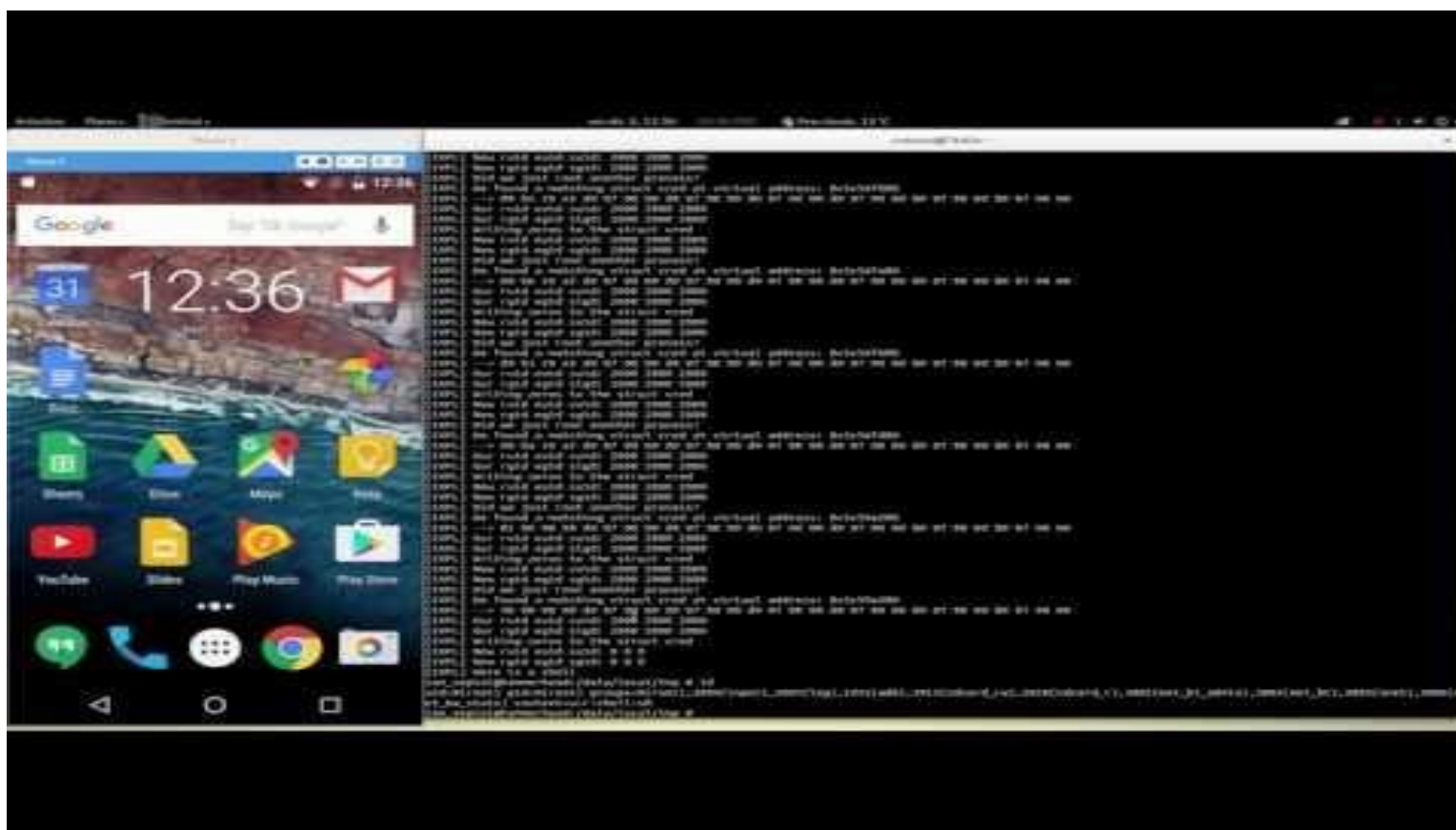
How did we get here?

- ❖ Computing industry demands more and faster storage with lower power consumption
- ❖ Ability of user to circumvent the caching system
    - ▪ `clflush` is an unprivileged instruction in x86
    - ▪ Other commands exist that skip the cache
- ❖ Availability of virtual to physical address mapping
    - ▪ **Example:** `/proc/self/pagemap` on Linux (not human-readable)

- ❖ Google patch for Android (Nov. 8, 2016)
    - ▪ Patched the ION memory allocator

More reading for those interested

❖ DRAMMER paper:
https://vvdveen.com/publications/drammer.pdf

❖ Google Project Zero:
https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

❖ First rowhammer paper:
https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

❖ Latest non-uniform, frequency-based exploit:
https://comsec.ethz.ch/research/dram/blacksmith/

❖ Wikipedia: https://en.wikipedia.org/wiki/Row_hammer