Virtual Memory III: The One with All the Examples

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

Wei Lin

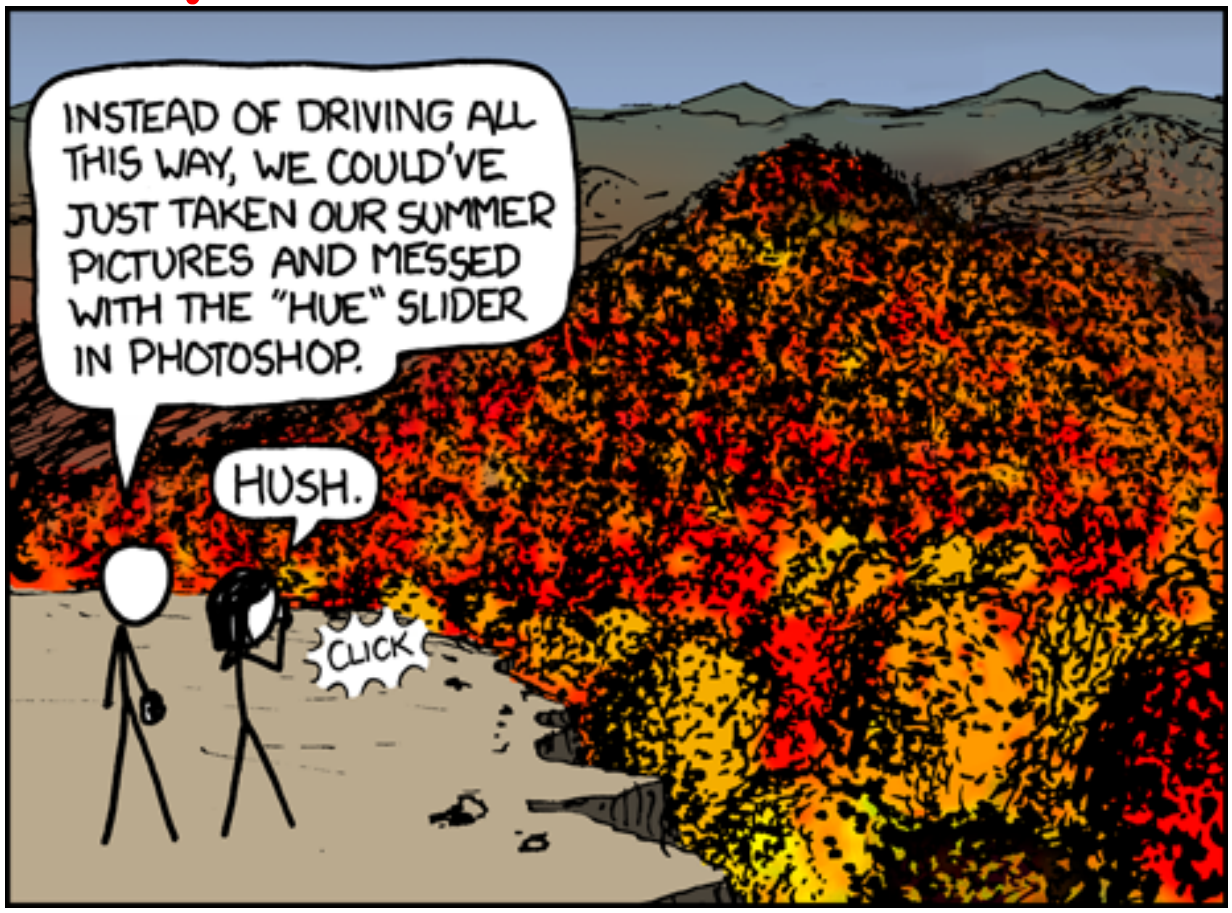
Daniel Snitkovsky

Luis Vega

Kory Watson

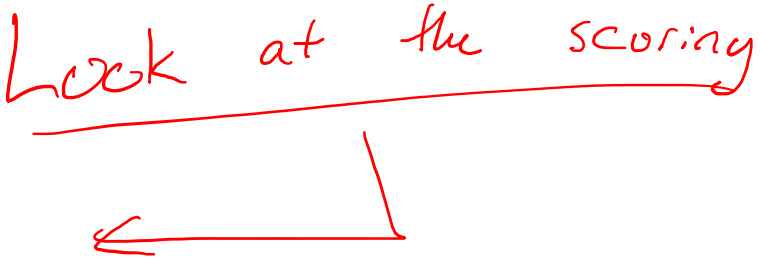
Ivy Yu

Running out of computer systems comics!



https://xkcd.com/648/

Administrivia



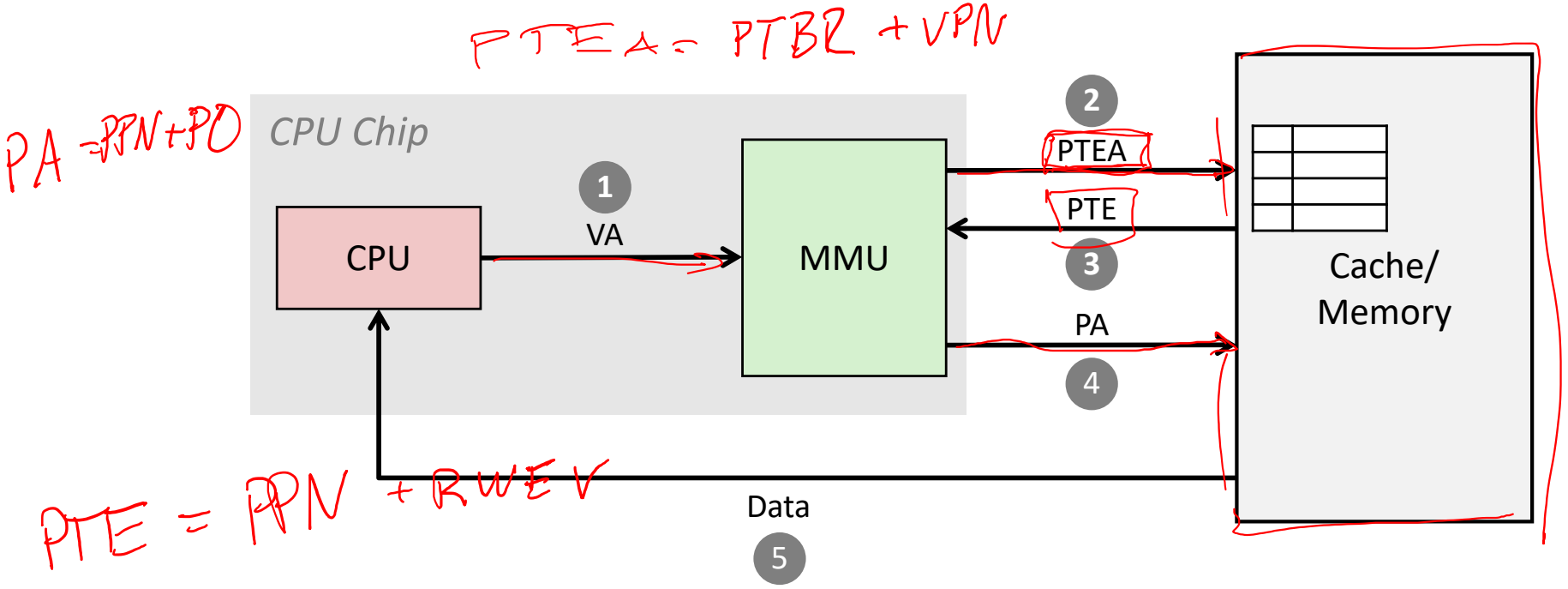
- Lab 4 due **today** (March 4)
- Lab 5, HW 5 coming soon

- Collaboration means the discussion of the *problem*
  - not the specifics of *your solution*

Quick Review

- What is VM useful for?
  isolation, protecting, easier to program, cache
- What do Page Tables map?
  VPN → PPN
- Where are Page Tables located?
  ~~PM~~ in physical memory
- How many Page Tables are there?
  1 for each process → isolation
- Can your process tell if a page fault has occurred?
  No... kinda
- True / ~~False~~: Virtual Addresses that are contiguous will always be contiguous in physical memory
- TLB stands for translation lookaside buffer (cache) and stores PT mappings

Address Translation: Page Hit



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

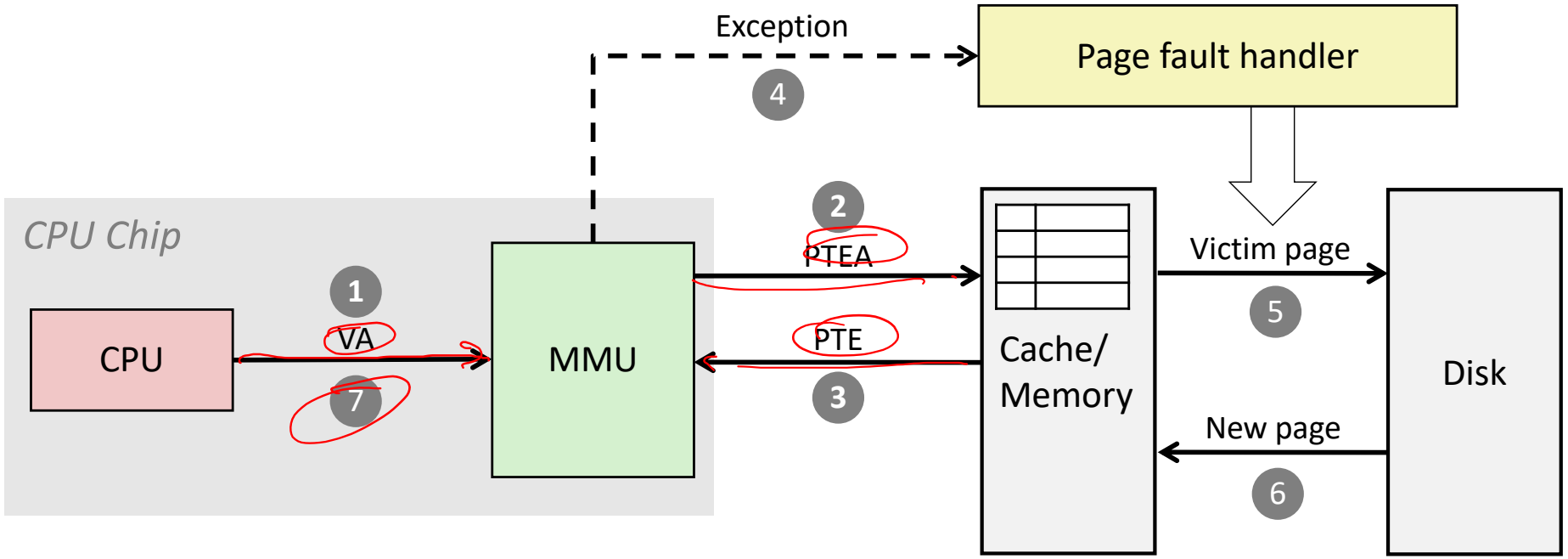
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

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

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

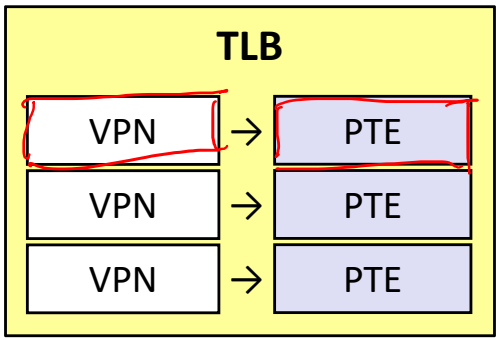
Hmm… Translation Sounds Slow

- The MMU accesses memory *twice* for a *page hit*
  - 1. get the PTE for translation
  - 2. 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

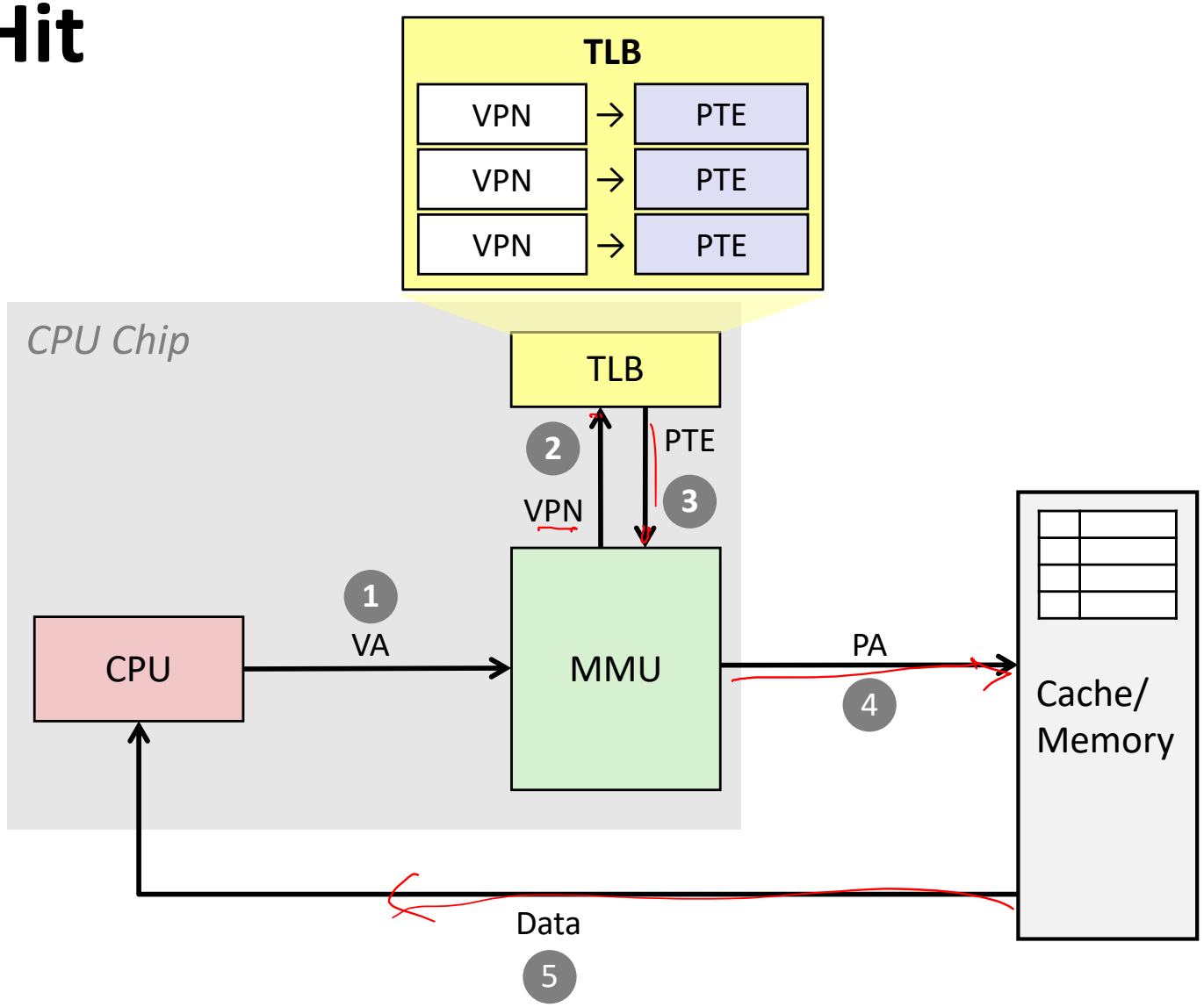
- *What can we do to make this faster?*
  - **Solution:** add another cache! 

Speeding up Translation with a TLB

- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory

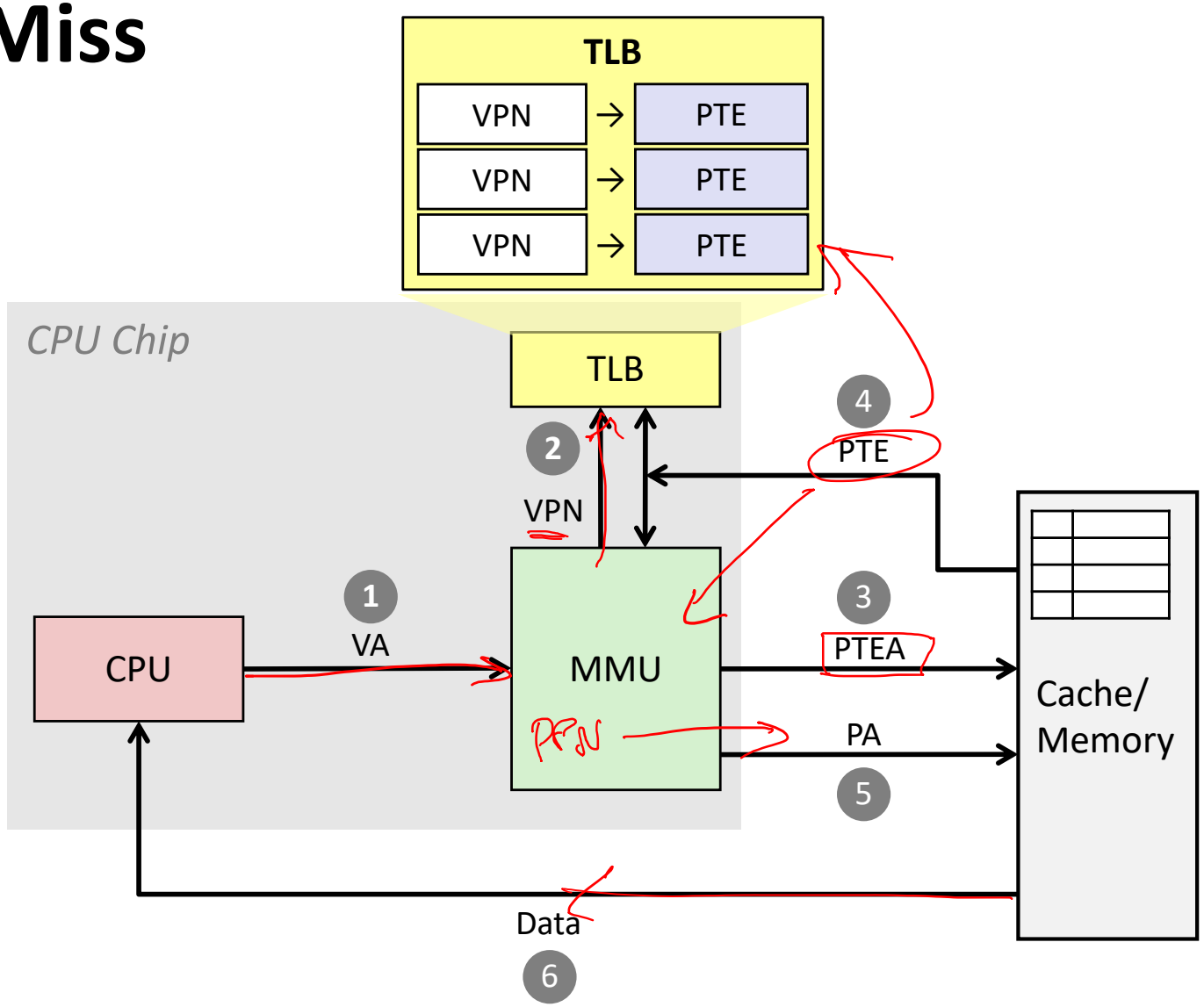


TLB Hit



- A TLB hit eliminates a memory access!

TLB Miss

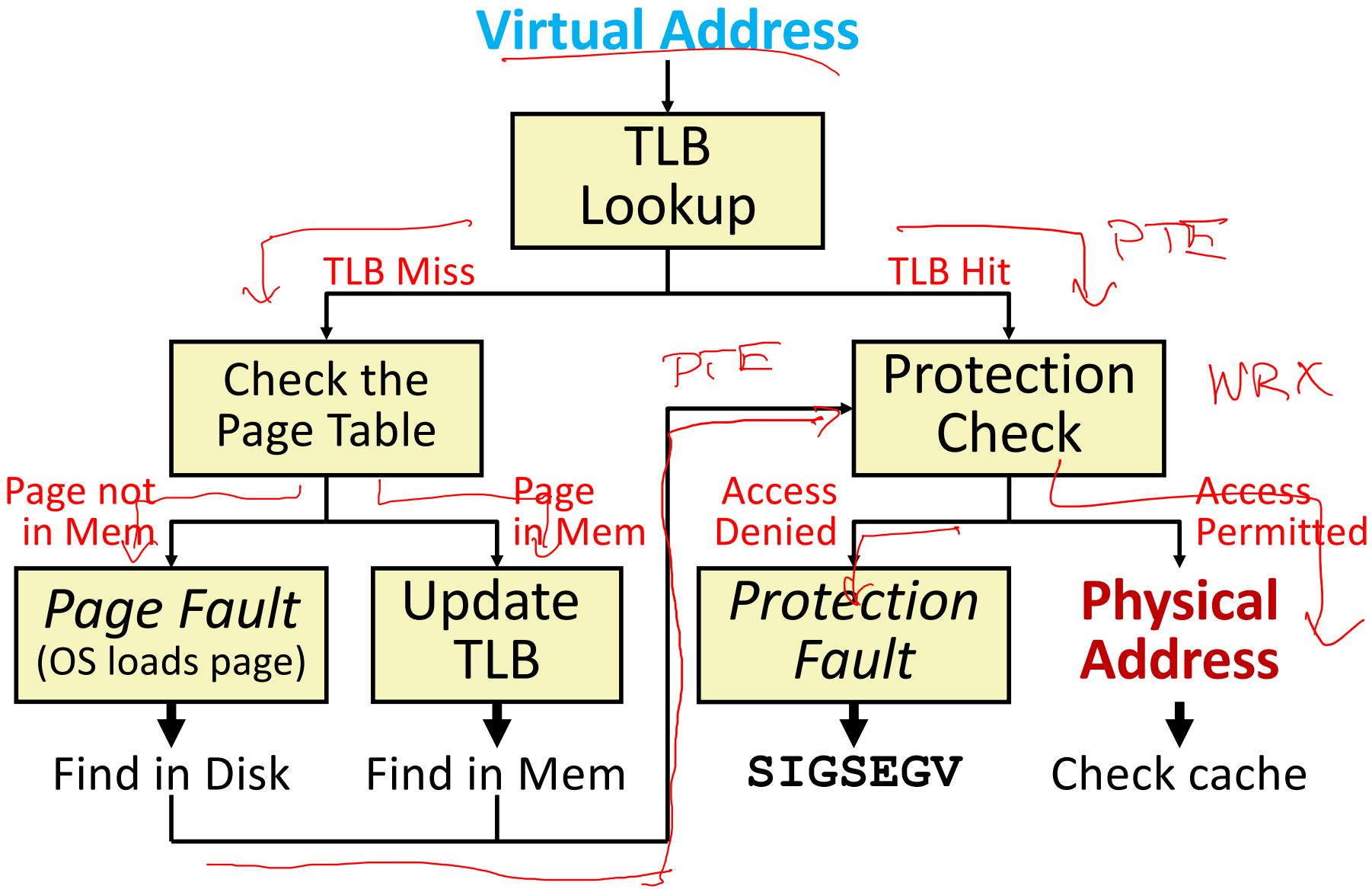


- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare

Fetching Data on a Memory Read

1) Check TLB
   - Input: VPN, Output: 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
   - Input: physical address, Output: 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

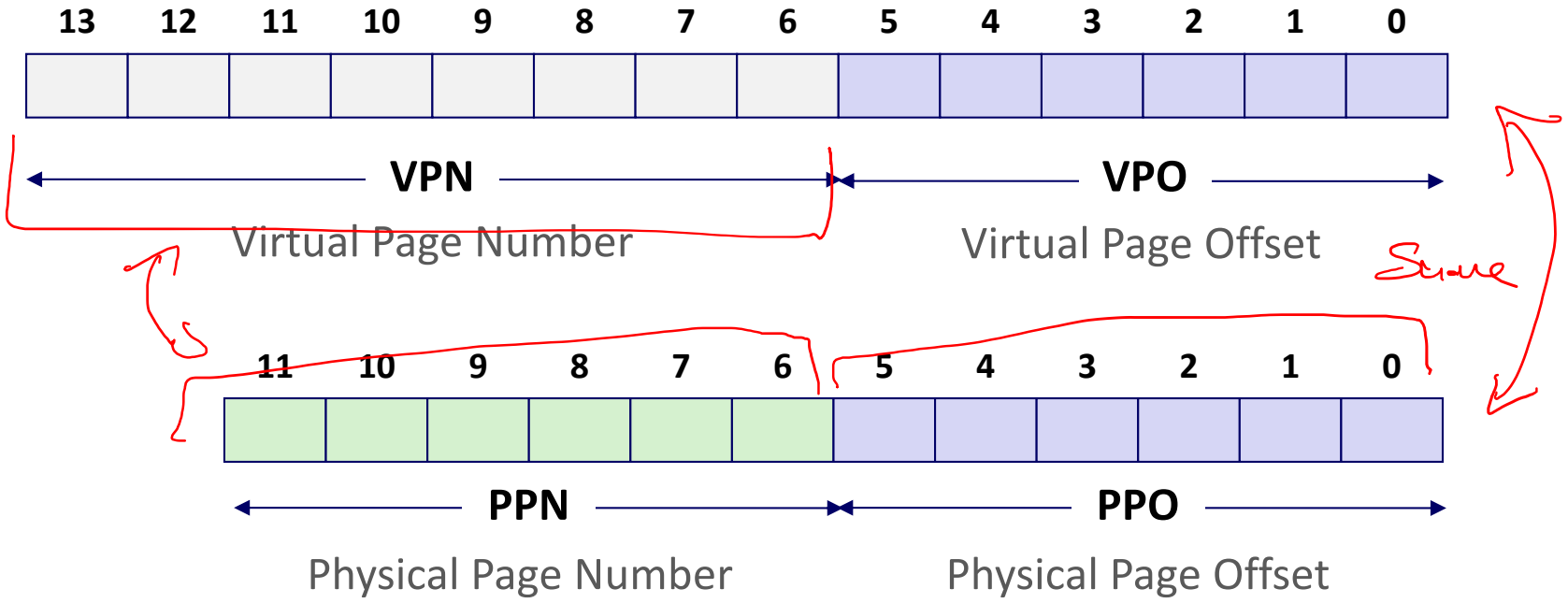


Context Switching Revisited

- 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

Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses → 16 kb VA space
  - 12-bit physical address → 4kb PA space
  - Page size = 64 bytes → 6 bits = PO



Simple Memory System: Page Table

14 bit VA → 6 bit PO

- Only showing first 16 entries (out of $2^8 = 256$)
  - **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

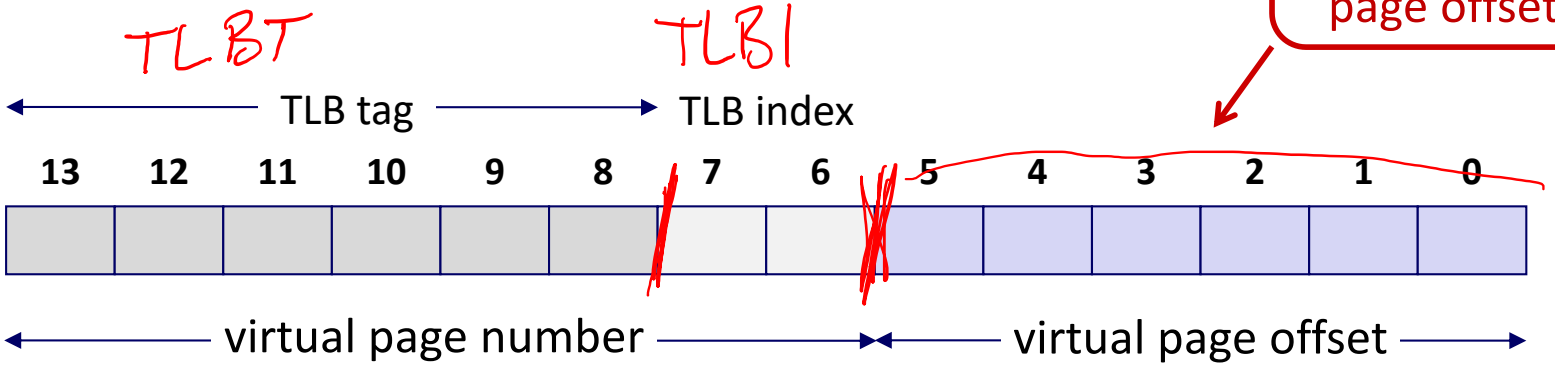
VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

16/4 = 4 sets

Why does the TLB ignore the page offset?



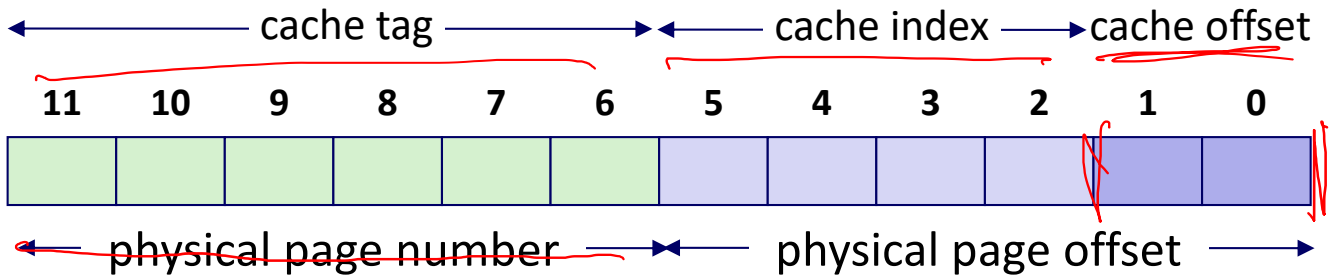
way 1 way 4

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

Circled #s refer to Memory Request Example #

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
③ 0	03	–	0	09	0D	1	00	–	0✗	07	02	1
④ 1	03	2D	1✓	02	–	0	04	–	0	0A	–	0
② 2	02	–	0	08	–	0	06	–	0	03	–	0✗
① 3	07	–	0	03	0D	1✓	0A	34	1	02	–	0

Page table (partial):

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

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

Cache:

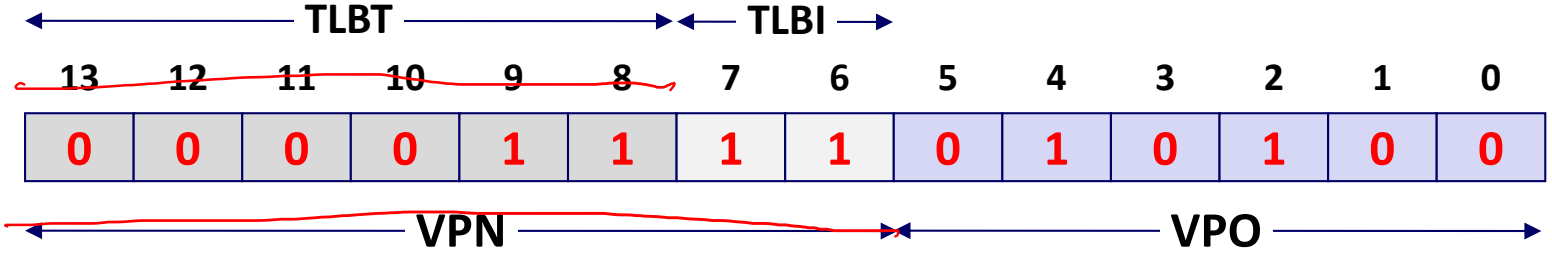
Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
① 5	0D✓	1✓	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	V	B0	B1	B2	B3
③ 8	24✗	1✓	3A	00	51	89
9	2D	0	–	–	–	–
④ 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	–	–	–	–

Memory Request Example #1

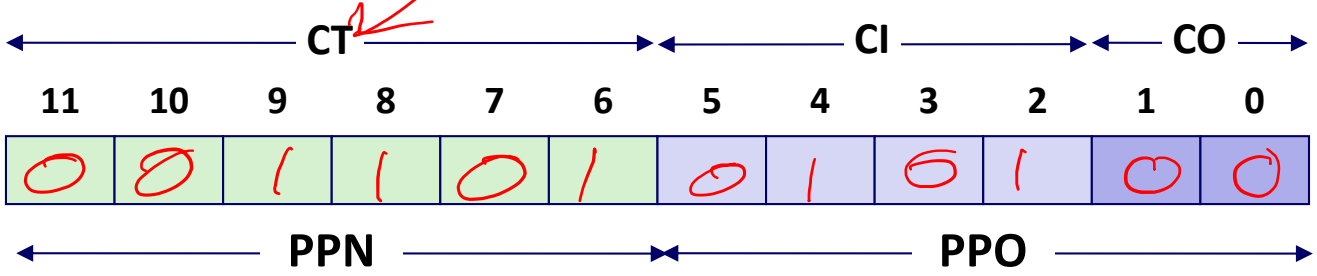
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x03D4



VPN 0F TLBT 3 TLBI 3 TLB Hit? Y Page Fault? NO PPN 0D

- Physical Address:

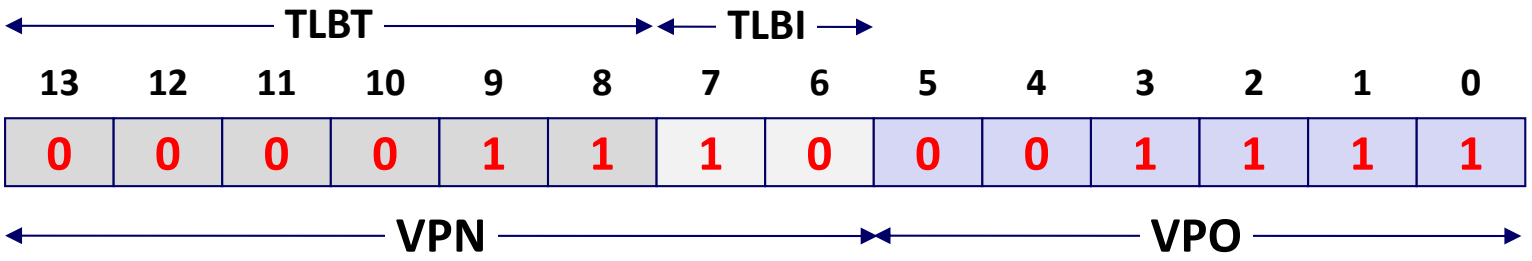


CT 0D CI 5 CO 0 Cache Hit? X Data (byte) 36

Memory Request Example #2

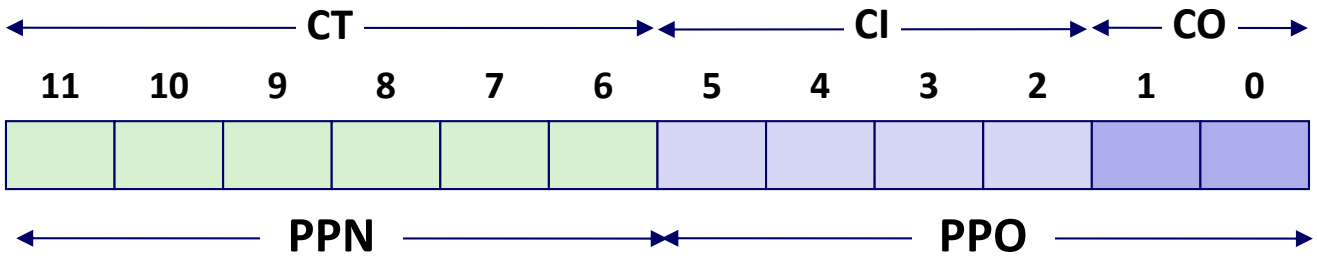
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x038F



VPN 0E TLBT 3 TLBI 2 TLB Hit? N Page Fault? Y PPN —

- Physical Address:

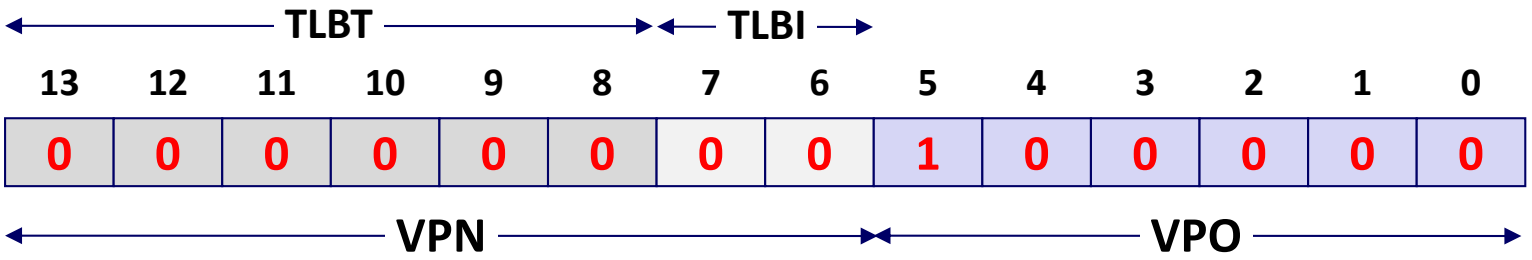


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #3

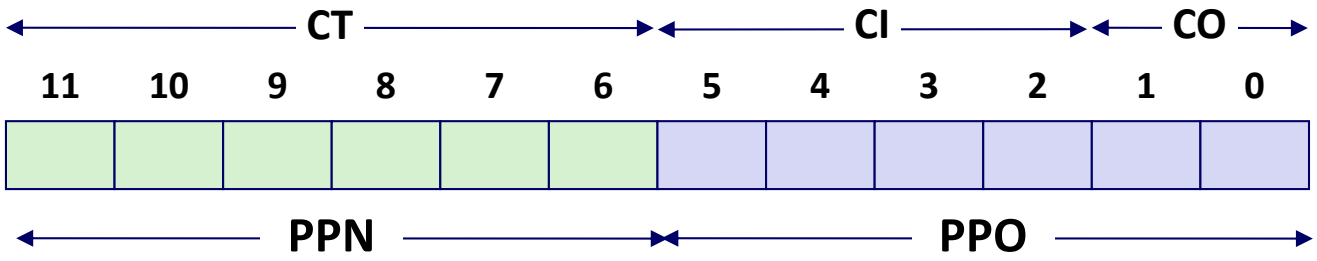
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x0020



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

- Physical Address:

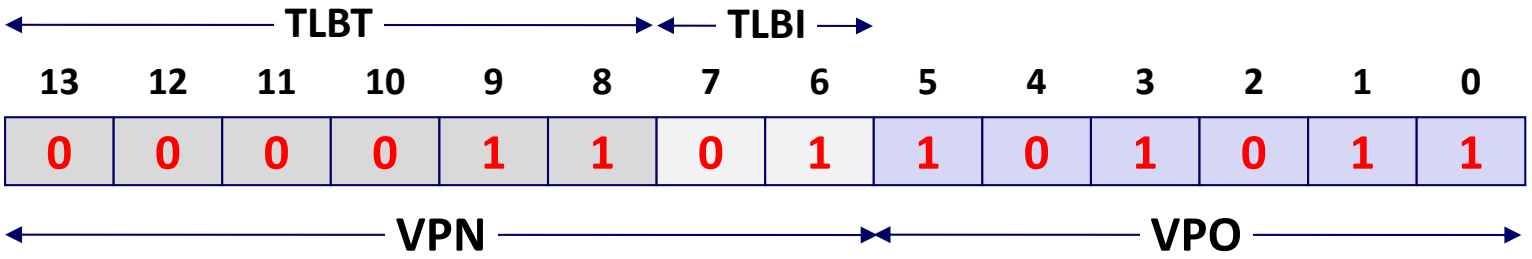


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #4

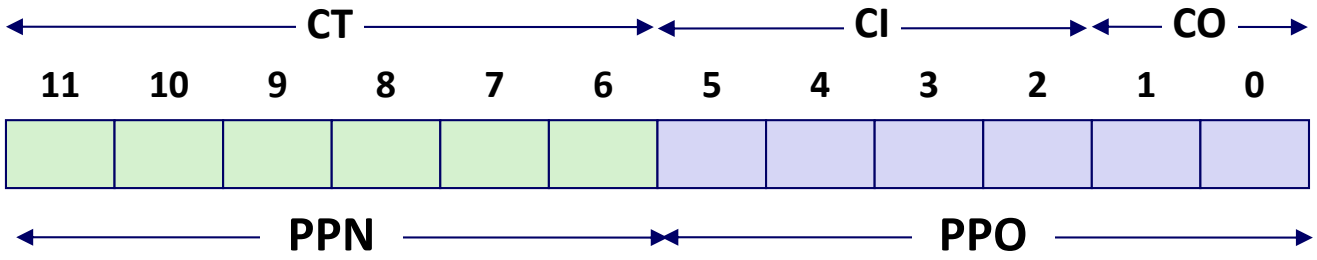
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x036B`



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

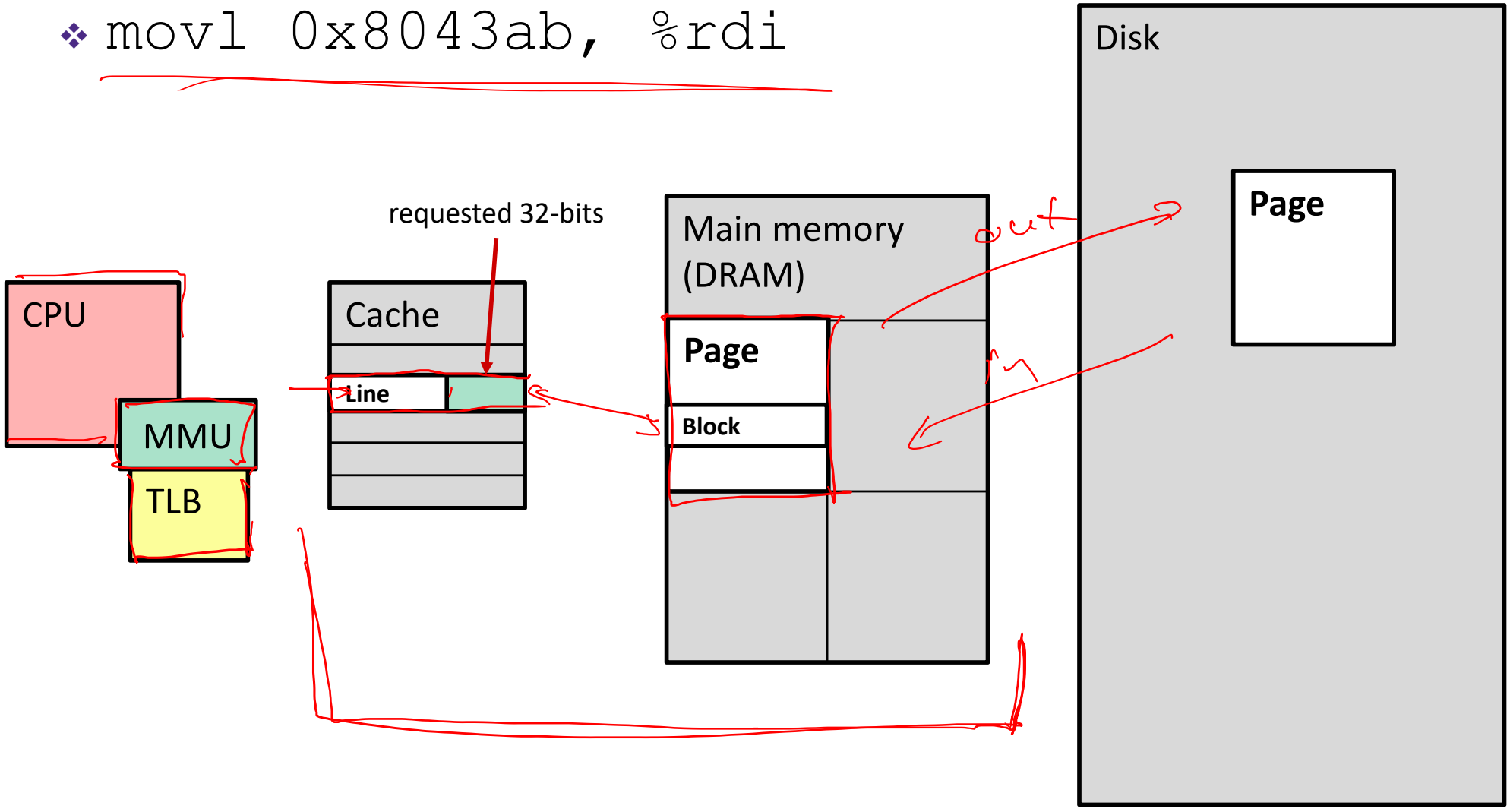
- Physical Address:



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

Memory Overview

- `movl 0x8043ab, %rdi`



Practice VM Question

- Our system has the following properties
  - 1 MiB of physical address space → 20 bits
  - 4 GiB of virtual address space → 32 bits
  - 32 KiB page size → 15 offset bits
  - 4-entry fully associative TLB with LRU replacement

↓ 1 set

a) Fill in the following blanks:

$2^{32}/2^{15} = 2^{17}$

_____ Entries in a page table

20 bits

_____ Minimum bit-width of PTBR

17

_____ TLBT bits

17 | 15
VPN | Offset

_____ Max # of valid entries in a page table

$2^{20}/2^{15} = 2^{5}$

Practice VM Question

- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

b) What is the largest stride (in bytes) between successive memory accesses (in the VA space)?

Practice VM Question

- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

c) Assuming all of `mat[]` starts on disk, what are the following hit rates for the execution of the for-loop?

_____ TLB Hit Rate _____ Page Table Hit Rate

Page Table Reality

This is extra (non-testable) material

- Just one issue… the numbers don't work out for the story so far!

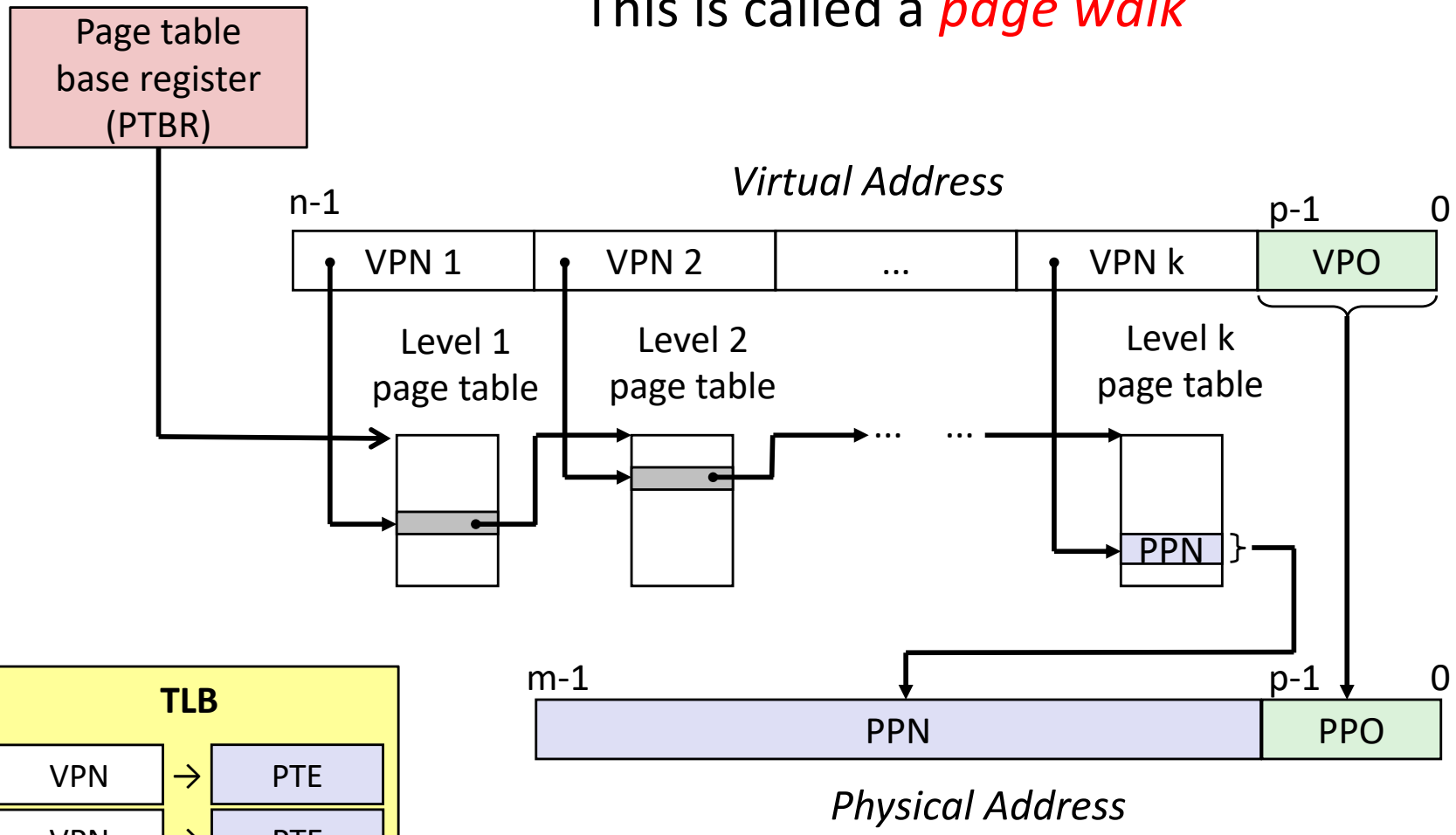
- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?

  - **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*



Multi-level Page Tables

This is extra (non-testable) material

- 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

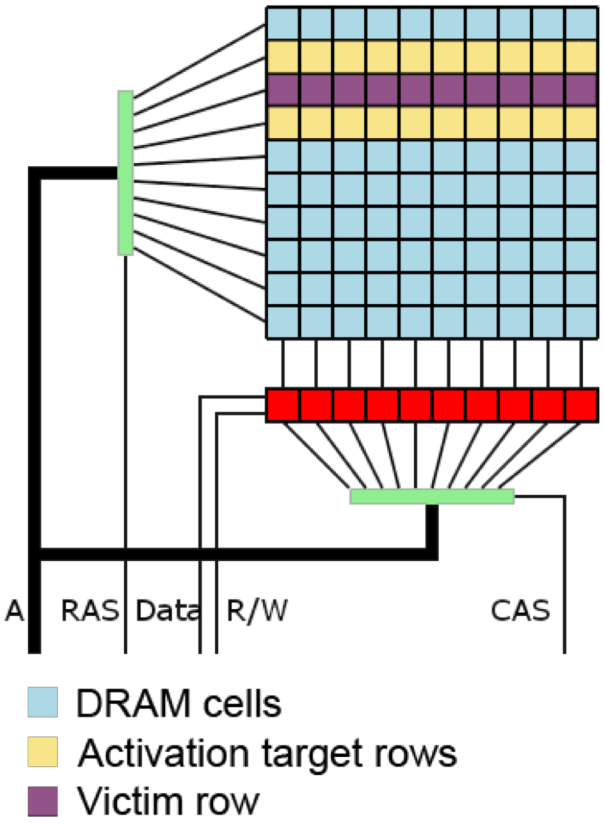
BONUS SLIDES

For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - **Recent:** Announced in October 2016; Google released Android patch on November 8, 2016
  - **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

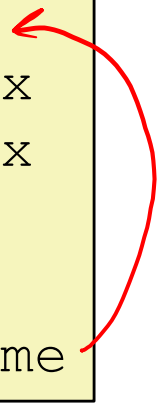


By Dsimic (modified), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38868341

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 row hammer paper: https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf
- Wikipedia: https://en.wikipedia.org/wiki/Row_hammer