Virtual Memory III

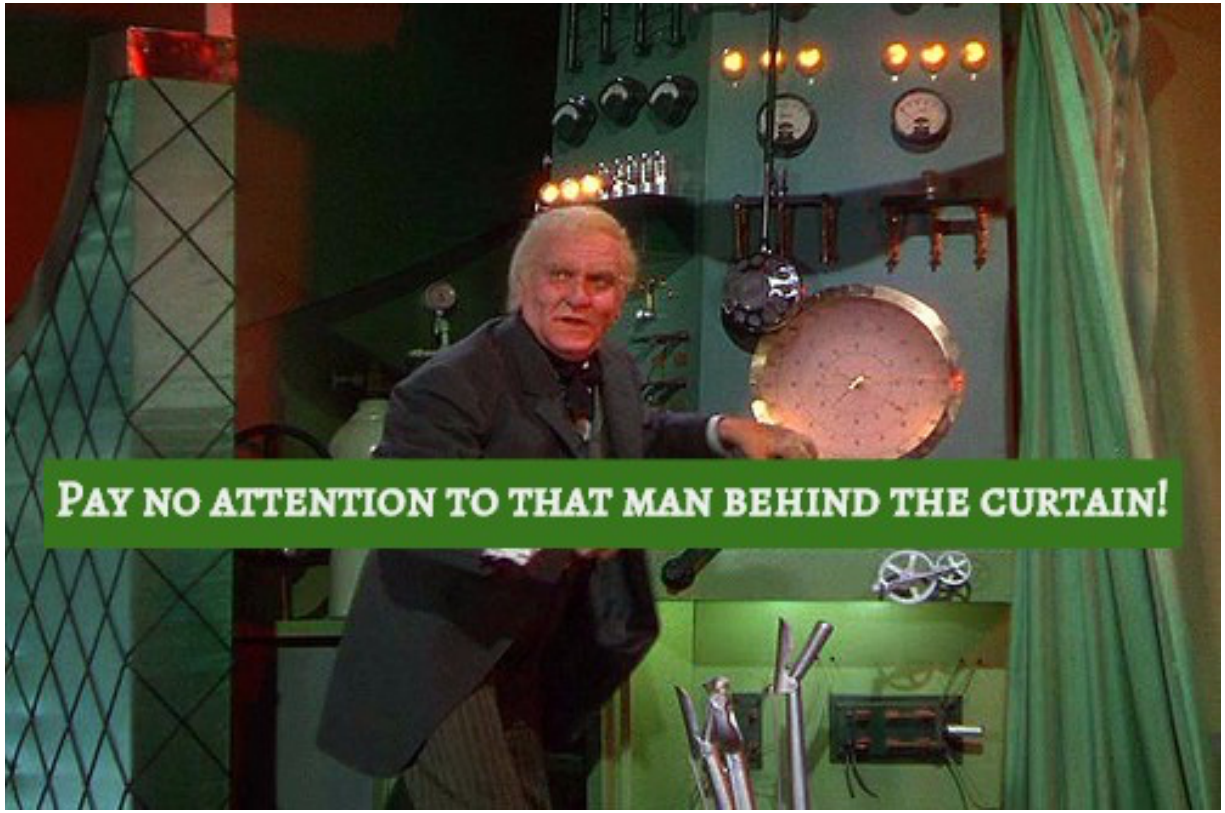
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

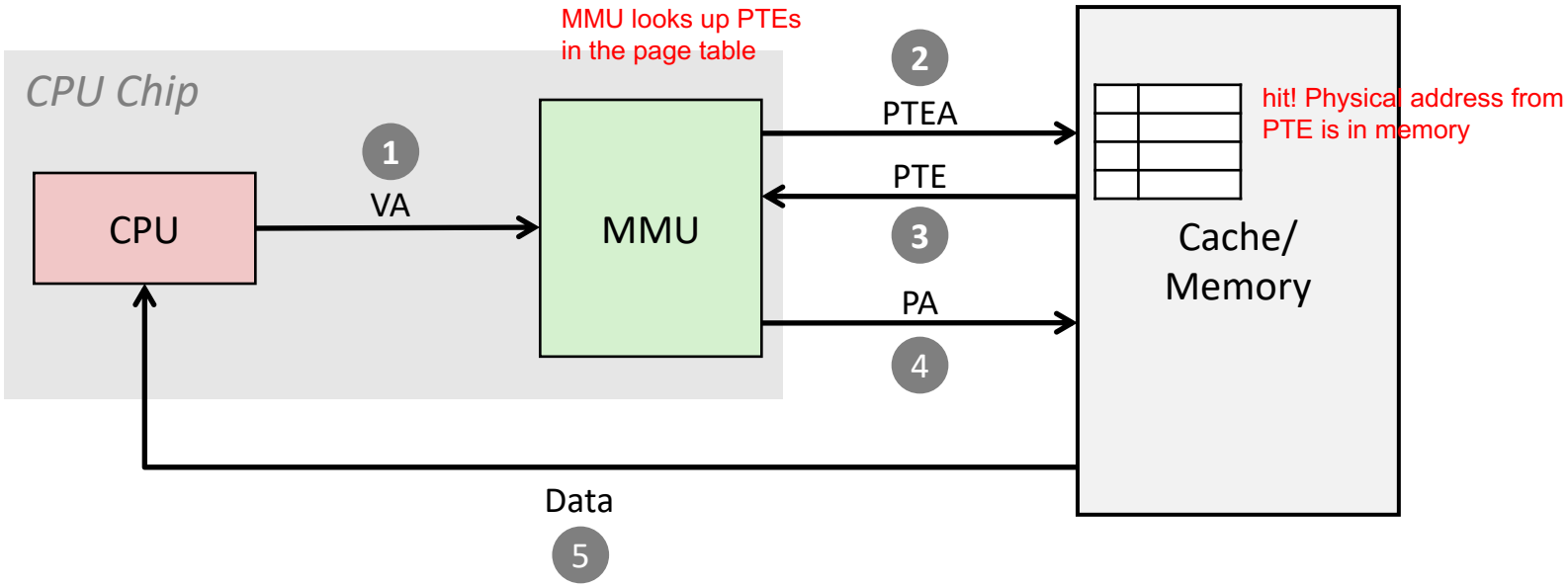
Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh



Administrivia

- Lab 4, due Fri (5/24)
- Homework 5 is out!
  - Processes and Virtual Memory
  - Due Friday, May 31
- Error on last week's section handout
  - Incorrect variable names for caches

Address Translation: Page Hit



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

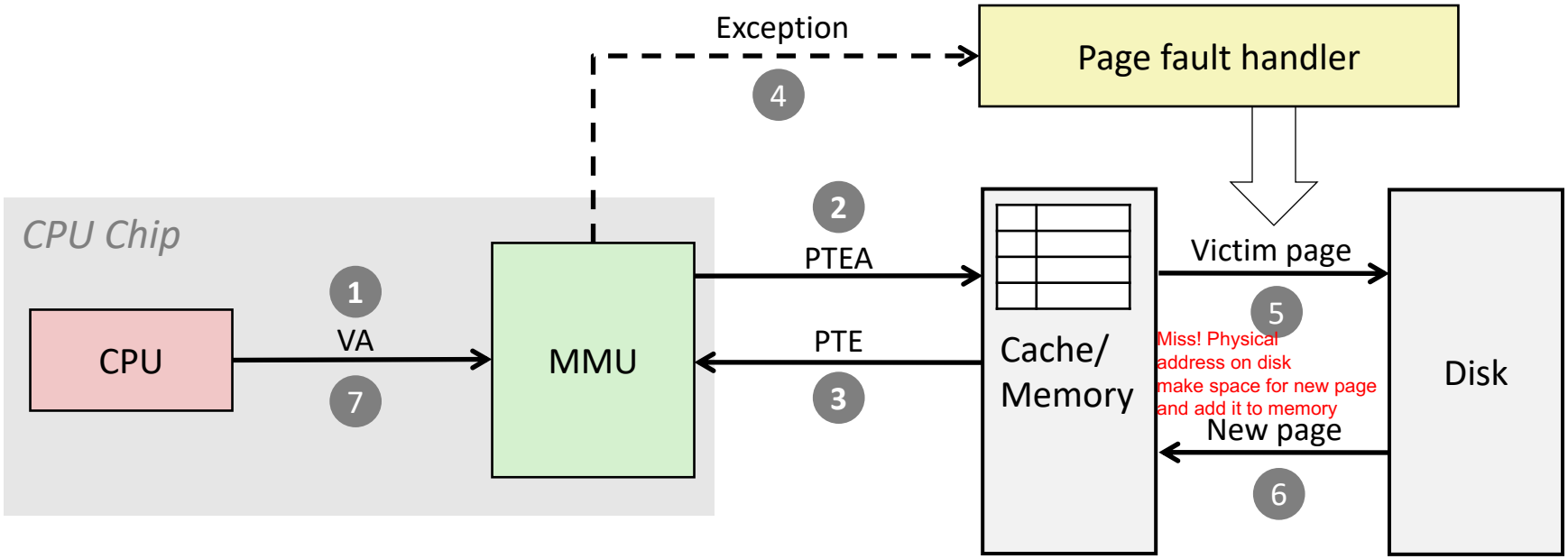
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

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)

e.g. if the page data doesn't match that found on disk, write the new data 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

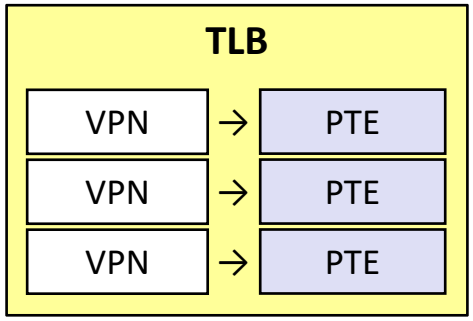
- The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- *What can we do to make this faster?*
  - "Any problem in computer science can be solved by adding another level of **indirection**." *– David Wheeler, inventor of the subroutine*
  - "And all of the new problems *that* creates can be solved by adding another **cache**." *- Sam Wolfson, inventor of this quote*

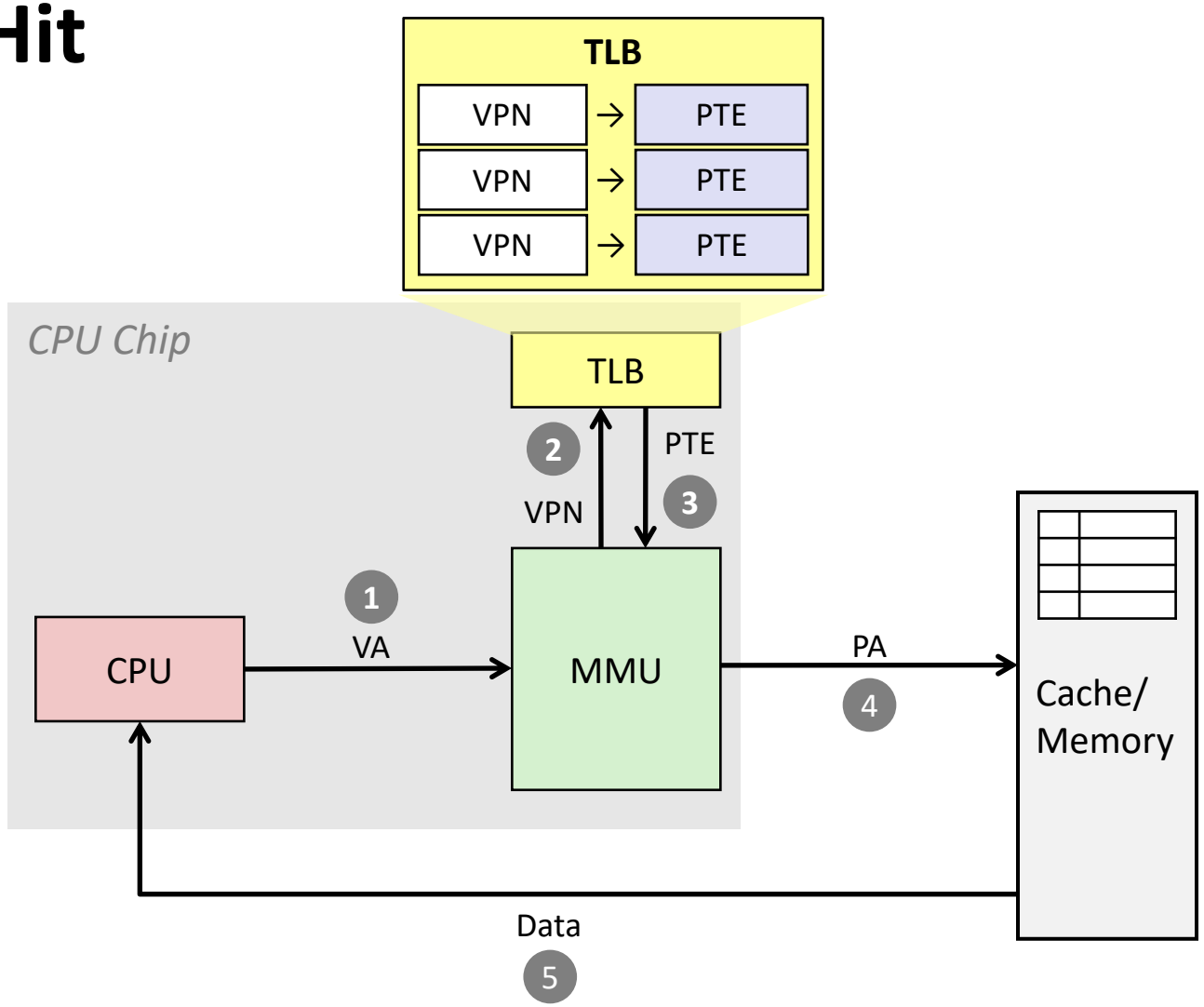
Speeding up Translation with a TLB

- *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

It is a subset of our page table that is stored in main memory.



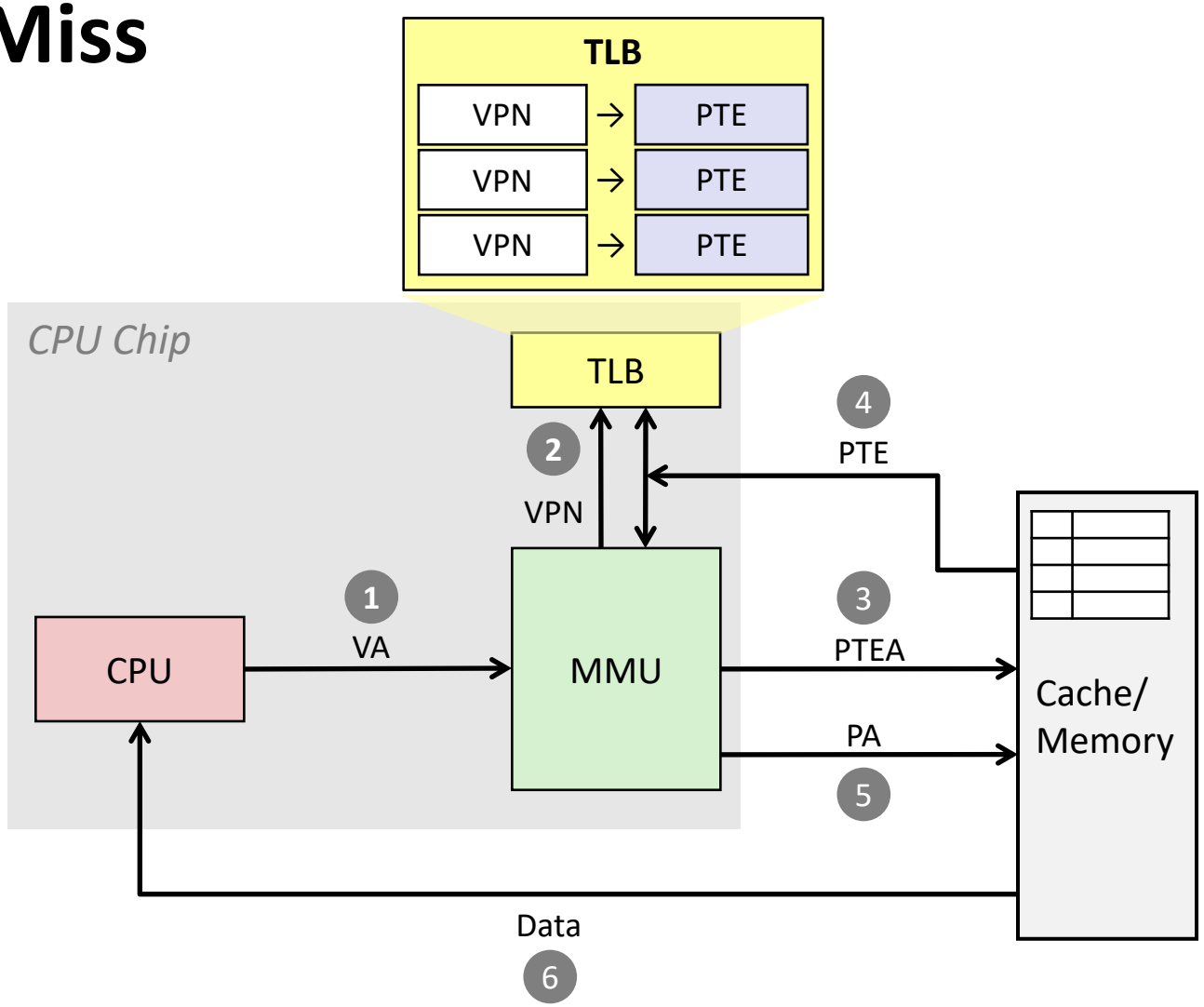
TLB Hit



- A TLB hit eliminates a memory access!

because we don't have to access the page table stored in main memory, just have to look up the physical address once we have it.

TLB Miss



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

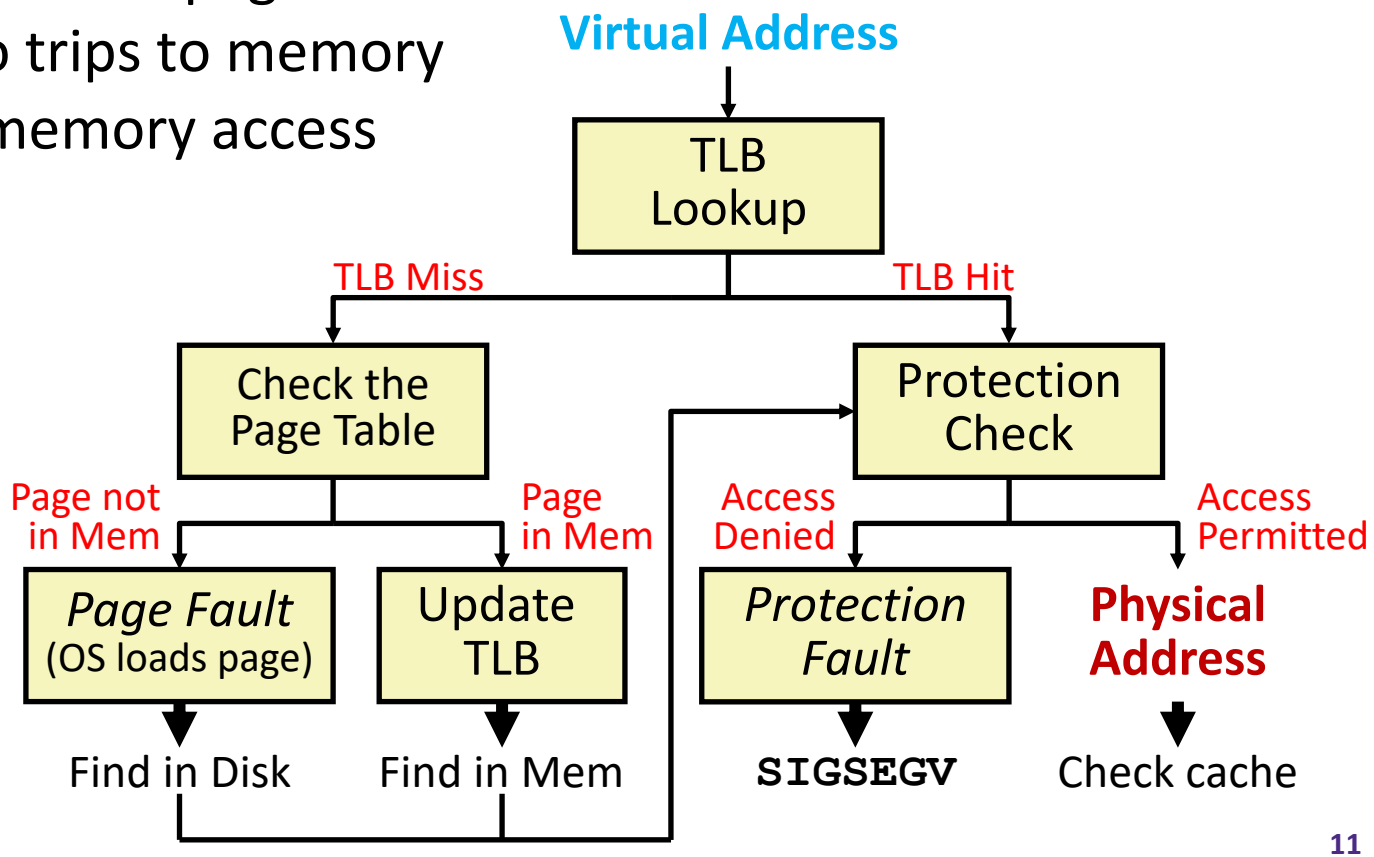
have to access the TLB, the page table in memory (due to miss), AND look up the physical address in memory

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

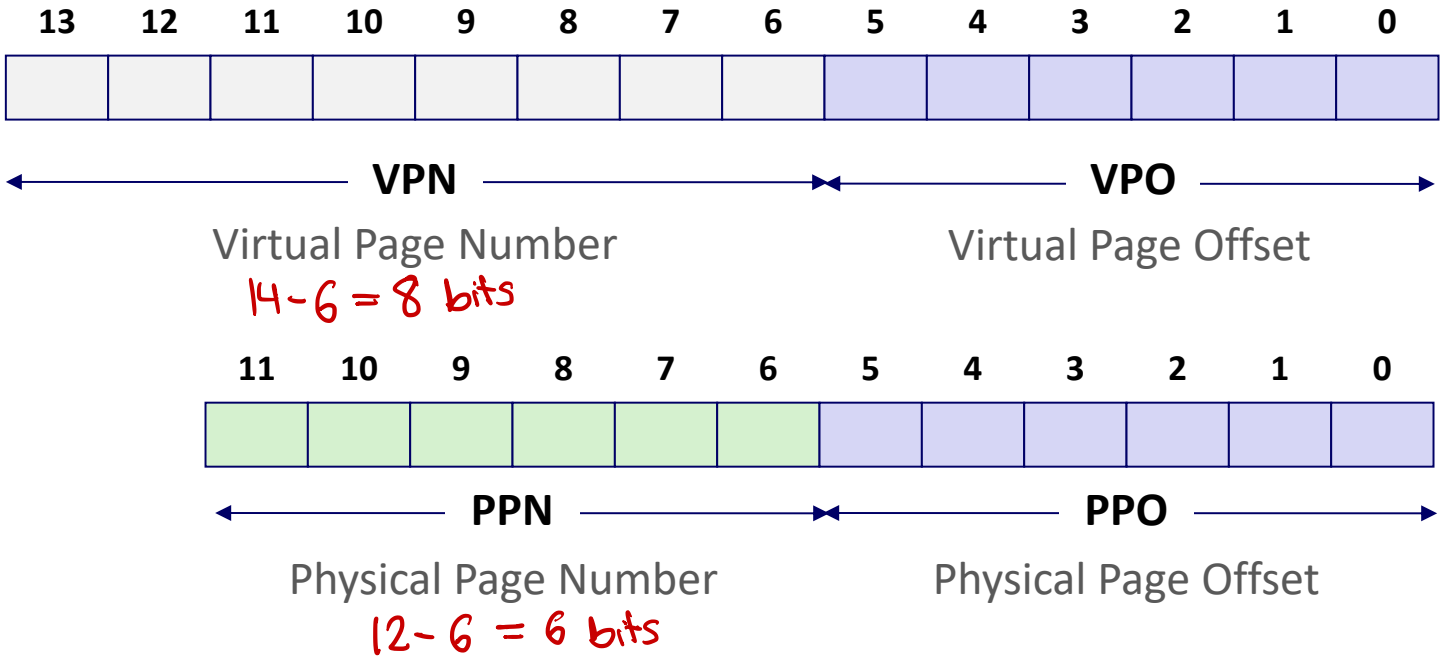
Address Translation

- VM is complicated, but also elegant and effective
  - Level of indirection to provide isolated memory & caching
  - TLB as a cache of page tables avoids two trips to memory for every memory access



Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses 2^{14} virtual addrs.
  - 12-bit physical address 2^{12} physical addrs.
  - Page size = 64 bytes $\log_2 64 = 6$ bit page offset



Simple Memory System: Page Table

$2^{(\text{VA size}) - (\text{VPO size})} = 2^{(\text{VPN size})}$

- Only showing first 16 entries (out of 2^8)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

e.g. the permission bits

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

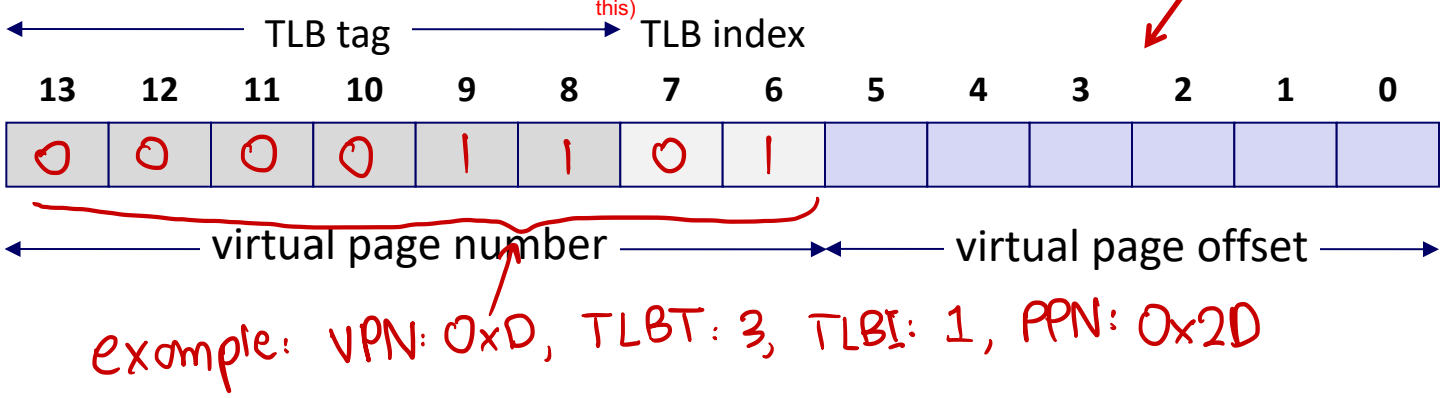
- 16 entries total
- 4-way set associative

VPO and PPO are the same thing, so no translation needed

Why does the TLB ignore the page offset?

4 sets each with 4 associativity

what set I am in (4 = 2^2 sets so need 2 bits to represent this)



set = 01 = 0x1
tag = 000011 = 0x03

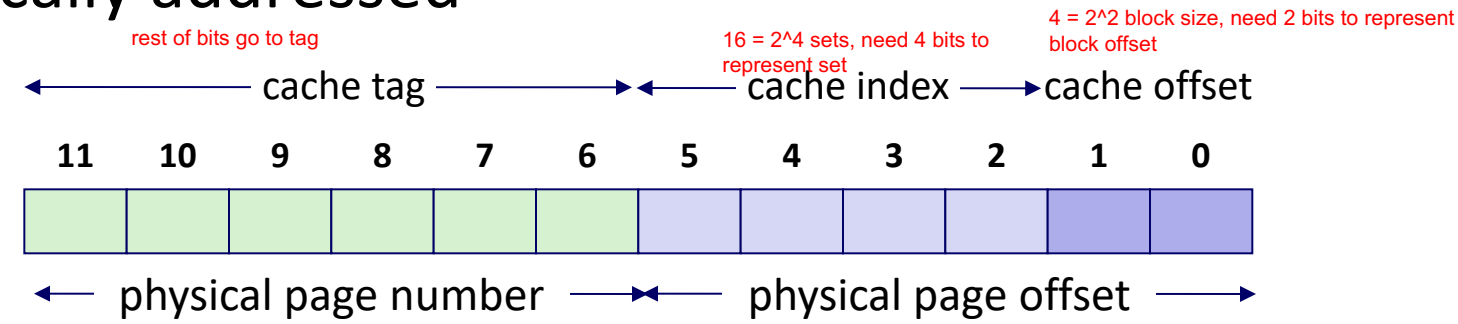
example: VPN: 0xD, TLBT: 3, TLBI: 1, PPN: 0x2D

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 e.g. 16 sets, directly mapped
  C/K = (num sets) * (set associativity)
- 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

only 16 out of 2^8 entries shown (page tables are huge, which is why we have the TLB for fast lookups)

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

TLB hit from next page

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:

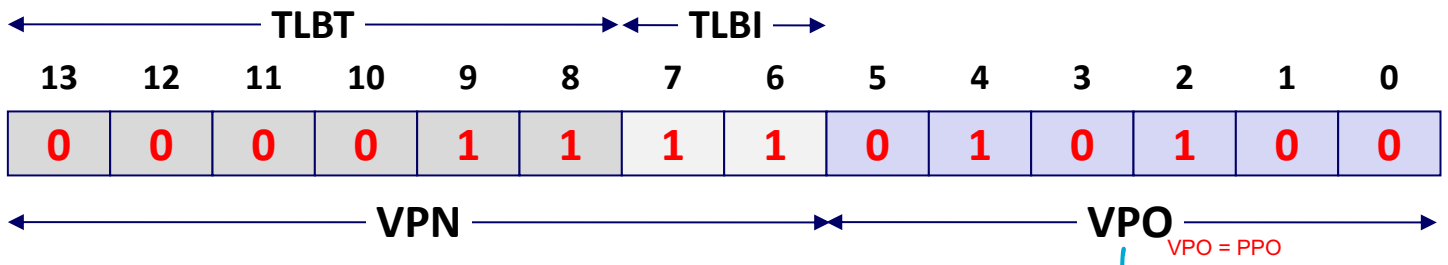
Index	Tag	V	B0	B1	B2	B3	Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	–	–	–	–	E	13	1	83	77	1B	D3
7	16	1	11	C2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

cache hit next page

Memory Request Example #1

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

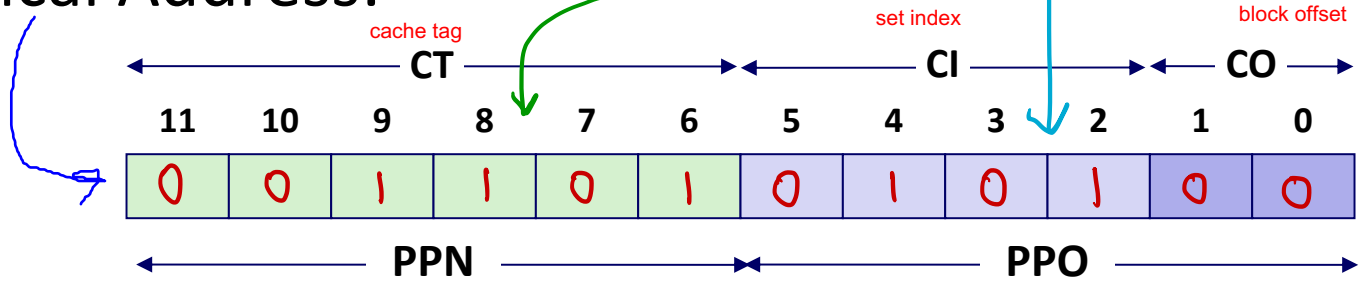
- Virtual Address: `0x03D4`



VPN 0xF TLBT 3 TLBI 3 TLB Hit? Y Page Fault? N PPN 0xD

TLB tag — TLB index — (see previous page blue circle)

- Physical Address:



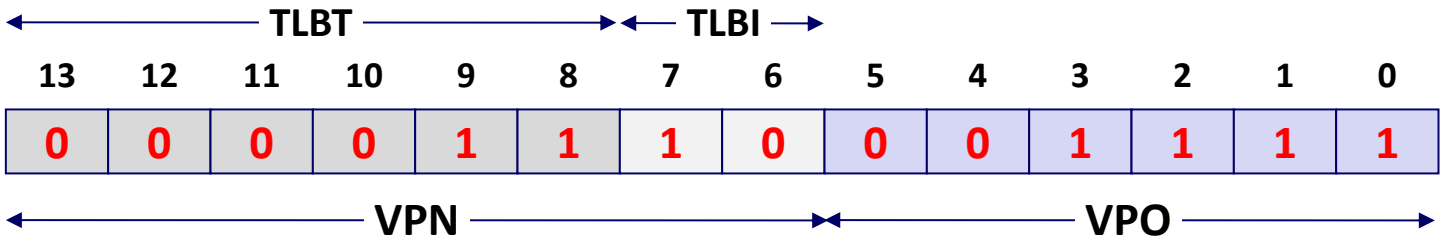
CT 0xD CI 5 CO 0 Cache Hit? Y Data (byte) 0x36

Cache tag — set index — block offset — (see previous page red circle)

Memory Request Example #2

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

- Virtual Address: `0x038F`

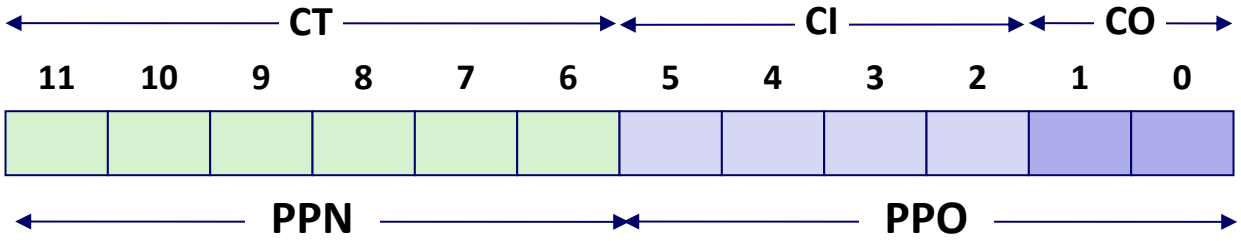


VPN 0xE TLBT 3 TLBI 2 TLB Hit? N Page Fault? Y PPN ¯_(ツ)_/¯

TLB set index
We can see this set is currently empty

have to load the data from disk into the cache; no physical just yet.

- Physical Address:

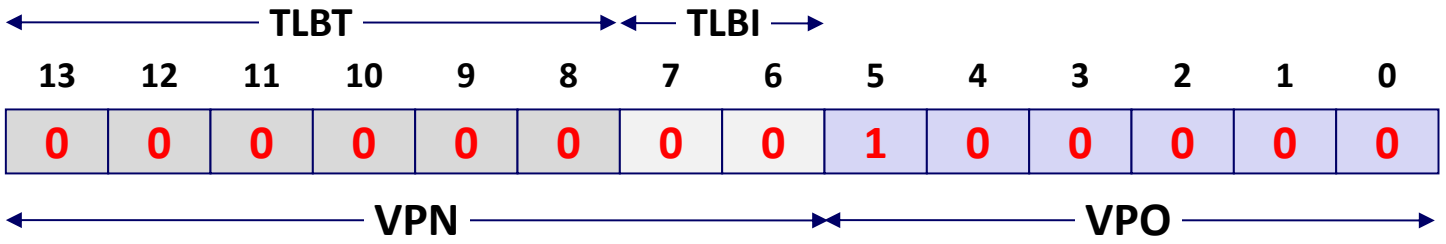


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #3

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

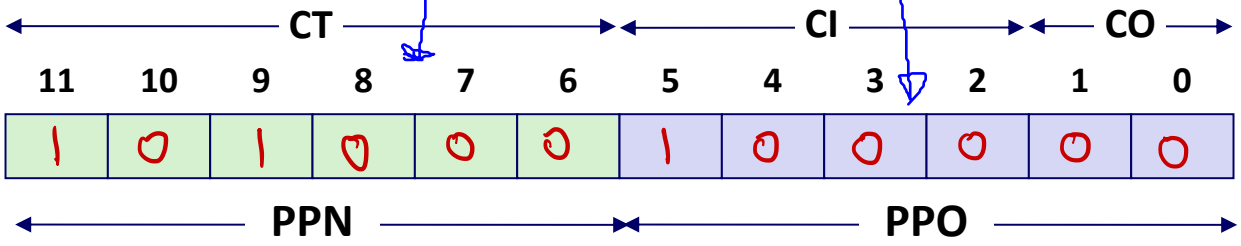
- Virtual Address: 0x0020



VPN 0 TLBT 0 TLBI 0 TLB Hit? N Page Fault? N PPN 0x28

from our TLB table

- Physical Address:



CT 0x28 CI 8 CO 0 Cache Hit? N Data (byte) ¯_(ツ)_/¯

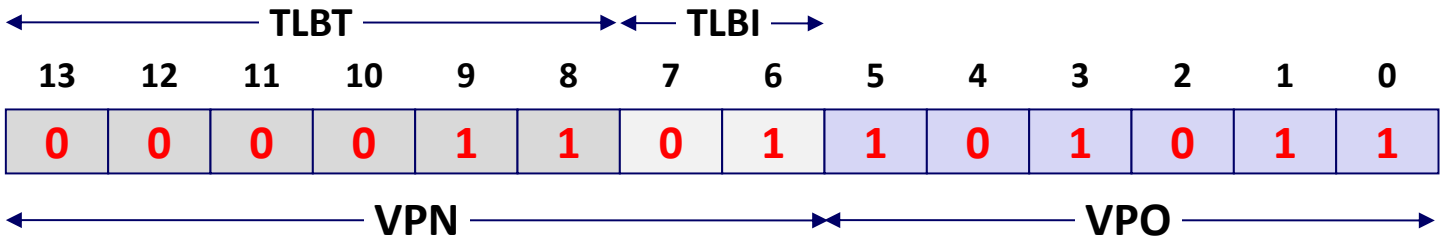
set index 8 in cache

Have to fetch the data from disk / lower memory level and move into cache.

Memory Request Example #4

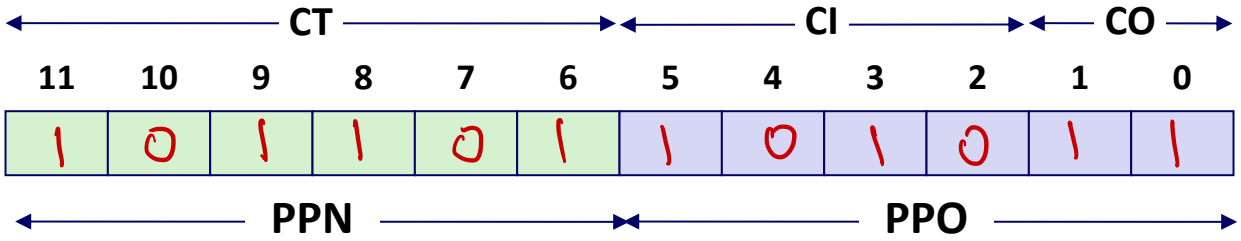
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x036B



VPN 0xD TLBT 0x3 TLBI 1 TLB Hit? Y Page Fault? N PPN 0x2D

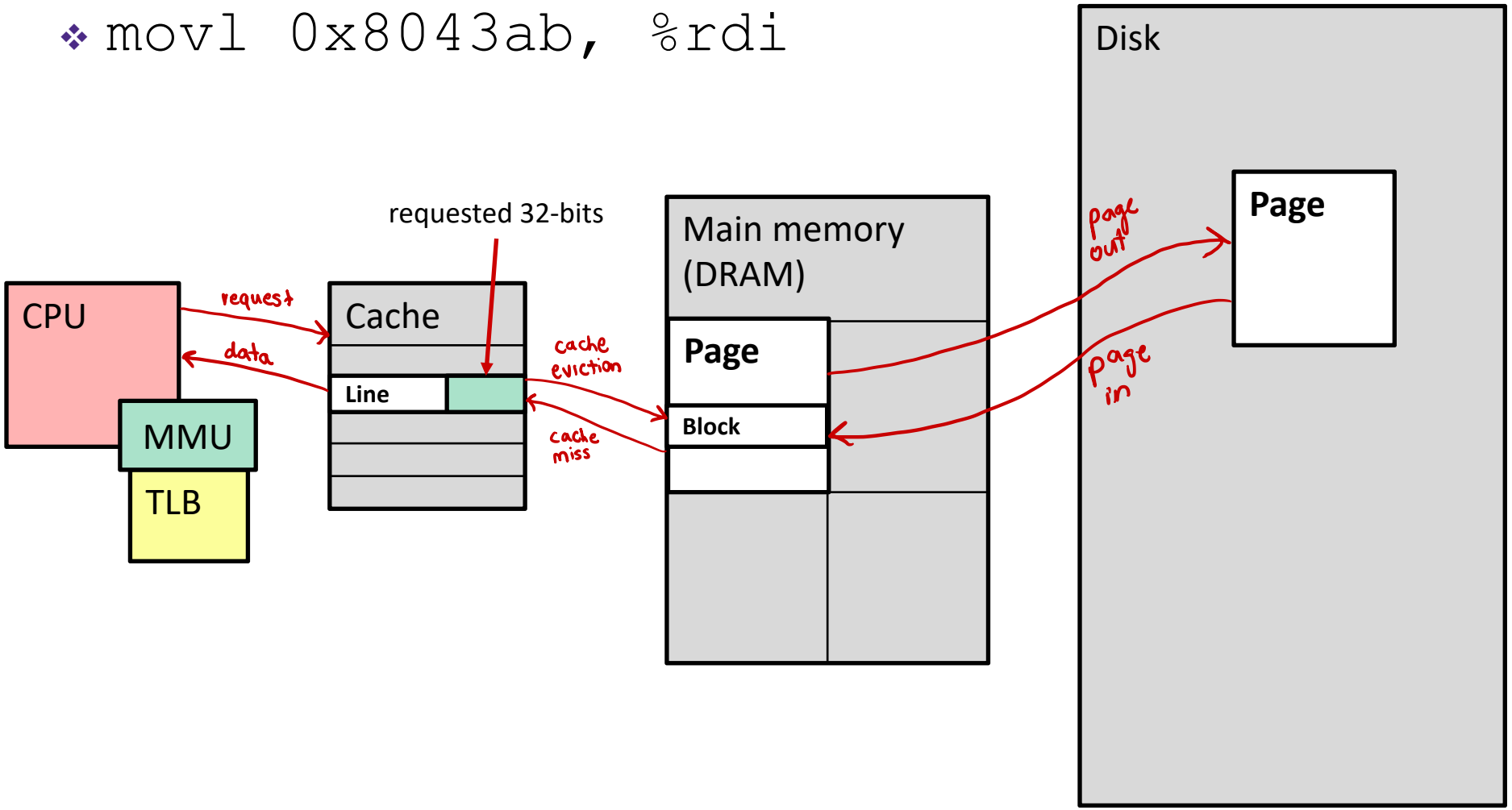
- Physical Address:



CT 0x2D CI 0xA CO 3 Cache Hit? Y Data (byte) 0x3B

Memory Overview

- `movl 0x8043ab, %rdi`



Page Table Reality

This is extra (non-testable) material

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

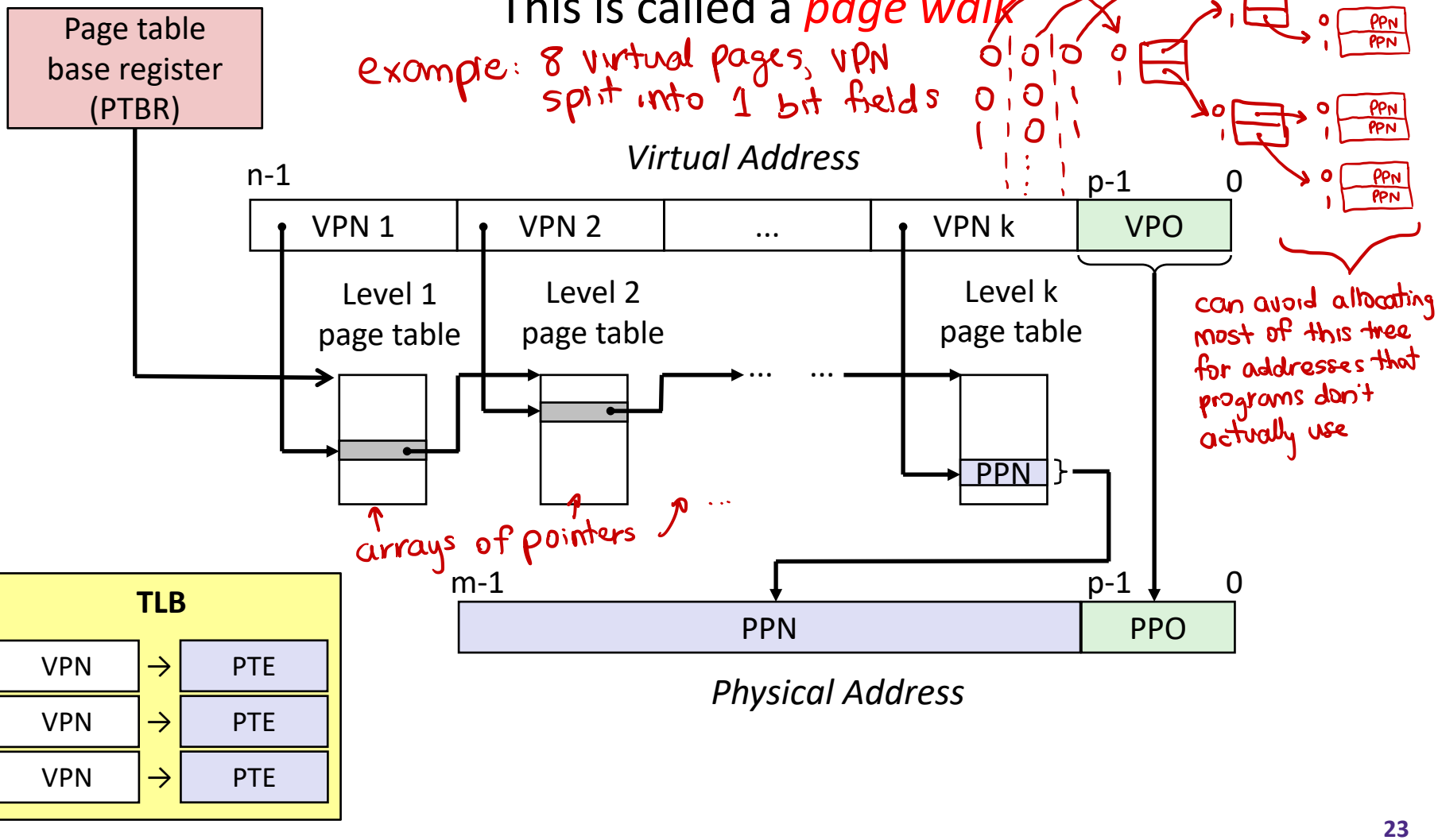
- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
    n = 64 bits, p = 13 bits, m = 33 bits
    page offset
  - How many page table entries is ~~that~~?
    1 entry per virtual page: $2^{n-p} = 2^{51}$ entries $\approx 3(2^{51}) = 2^{51} + 2^{52}$ bytes per page table!
    VPN bits = n-p = 51 bits --> 2^51 virtual pages
  - About how long is each PTE?
    PPN width (m-p = 20 bits) + management bits (R, W, X, etc.) = 23 bits ≈ 3 bytes
  - **Moral:** Cannot use this naïve implementation of the virtual→physical page mapping – it's *way* too big

    e.g. we cannot just store the entire page table in main memory

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

Practice VM Question

- Our system has the following properties
  - 1 MiB of physical address space $m = 20$ bits
  - 4 GiB of virtual address space $n = 32$ bits
  - 32 KiB page size $p = 15$ bits
  - 4-entry fully associative TLB with LRU replacement (1 set)

a) Fill in the following blanks:

2^{17} / 2^{n-p} Entries in a page table

20 Minimum bit-width of page table base register (PTBR) ← physical address

17 TLBT bits (VPN = TLBT + TLBI, 1 set so TLBI = 0)

2^5 / 2^{m-p} Max # of valid entries in a page table (# of pages in phys. mem.)

Practice VM Question

start addr. of mat at page offset 0

- One process uses a page-aligned 2048 × 2048 *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;
```

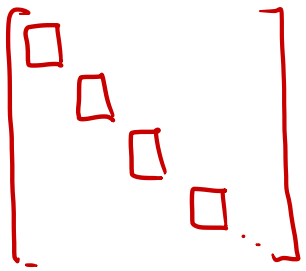
updating diagonals

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

i	index accessed
0	0
1	2049
2	2*2049
⋮	⋮

stride is always 2049 ints

= 8196 bytes



Practice VM Question

page size = 32 KB = 2^{15} B

- One process uses a page-aligned 2048 × 2048 *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;
```

2^{11} ints = 2^{13} B

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

_____ TLB Hit Rate

access pattern: single write to index, never revisited, access each row exactly once

each page holds $2^{15}/2^{13}$ = 4 rows of matrix

each page: MHHH

0% _____ Page Table Hit Rate

PT only accessed on TLB miss, because mat starts on disk each first access to a page is a page fault.

BONUS SLIDES

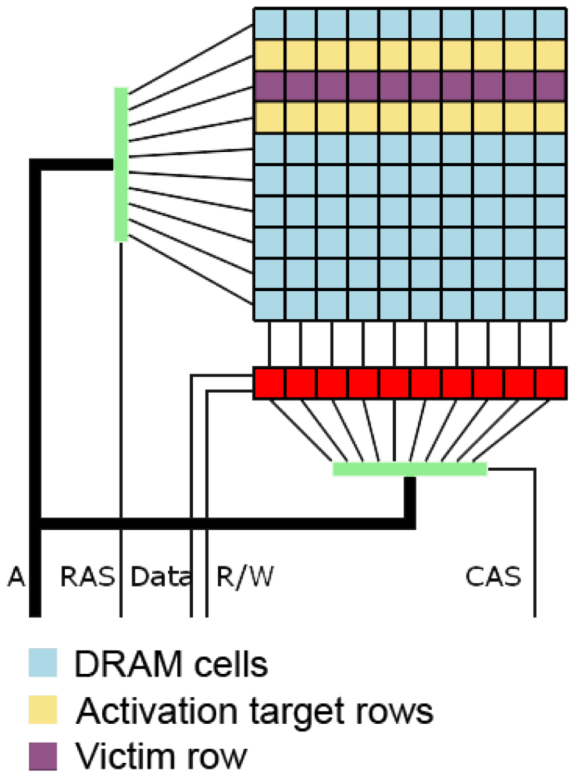
For Fun: DRAMMER Security Attack

- Why are we talking about this?
  - **Recent(ish):** 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

HARDWARE vulnerability

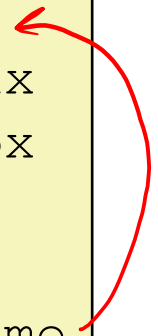


By Dsimic (modified), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38868341

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