Virtual Memory III

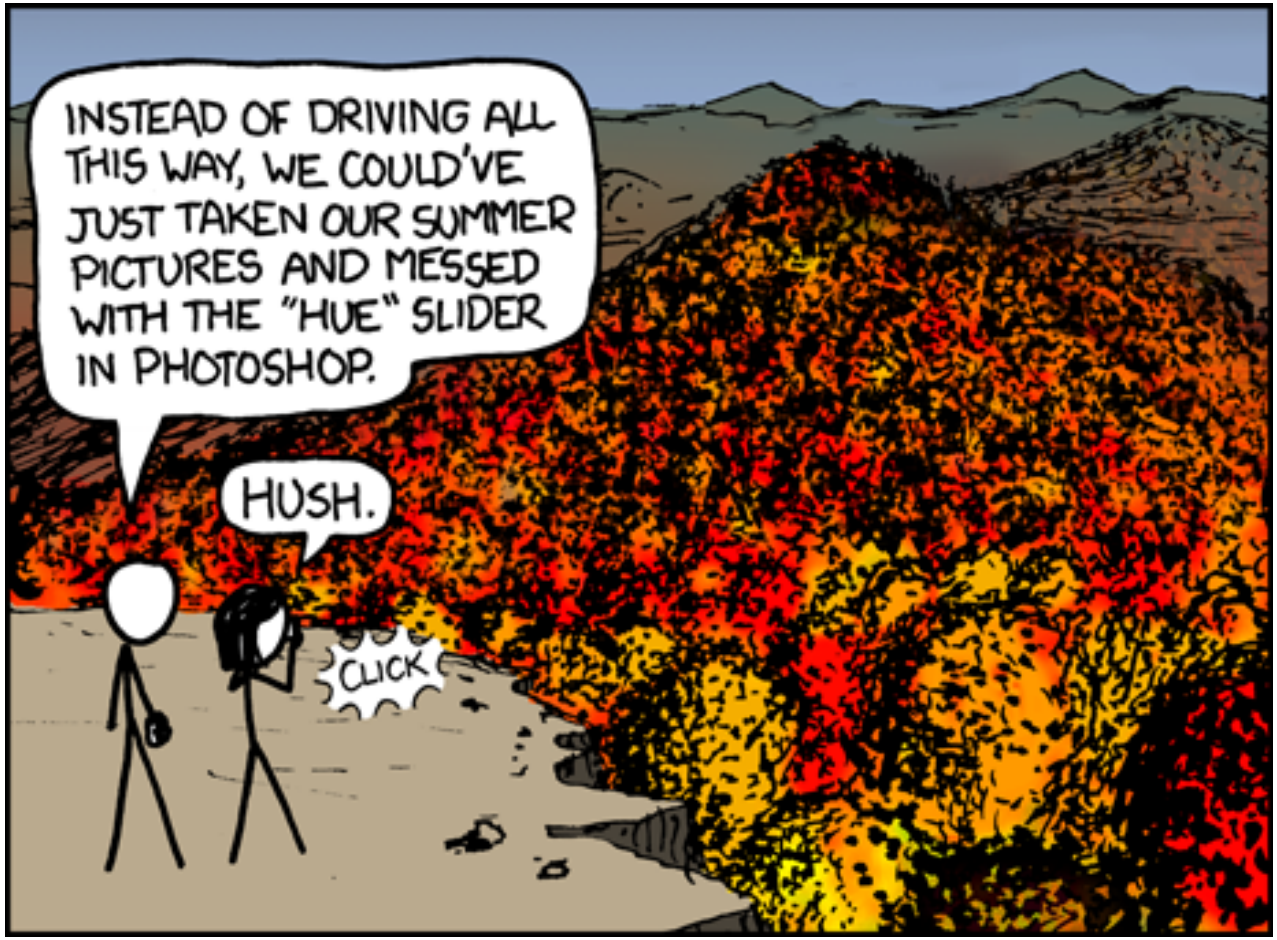
CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan



https://xkcd.com/648/

Administrivia

- Lab 4 due Monday (11/26)
- Homework 5 due next Friday (11/30)

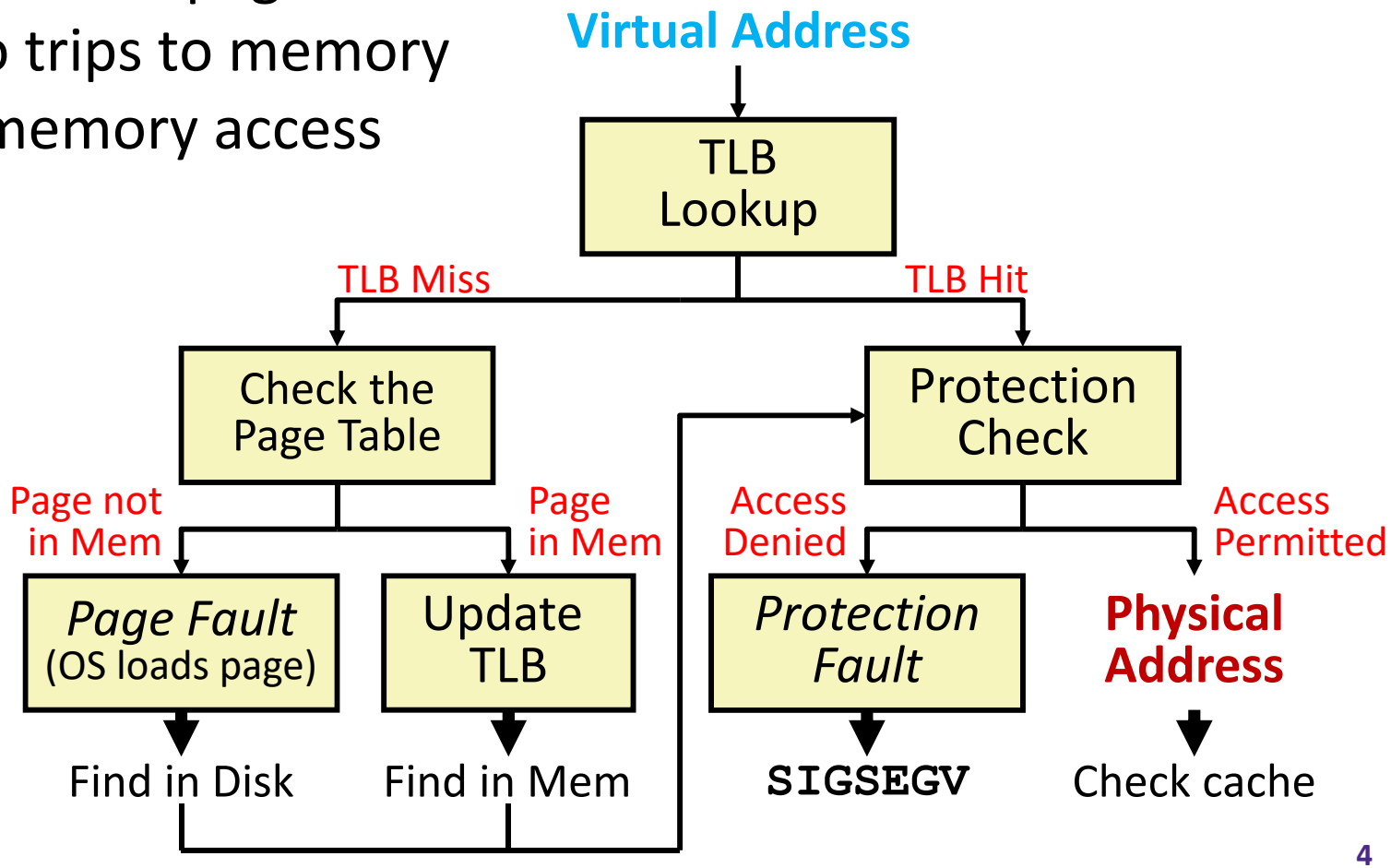
- "Virtual section" on virtual memory released
  - 3 PDFs: VM overview, worksheet, and solutions
  - Linked in the code section of today's lecture
  - See Piazza post for links and videos

Quick Review

- What do Page Tables map?
  VPN → PPN or disk address
- Where are Page Tables located?
  physical memory
- How many Page Tables are there?
  one per process
- Can your process tell if a page fault has occurred?
  No. MMU/OS throws page fault; process just waits for data
- True / False: Virtual Addresses that are contiguous will always be contiguous in physical memory
  page boundary: x | x+1
  pages can be mapped to any slot in physical mem
- TLB stands for translation lookaside buffer and stores page table entries
  British for "cache"

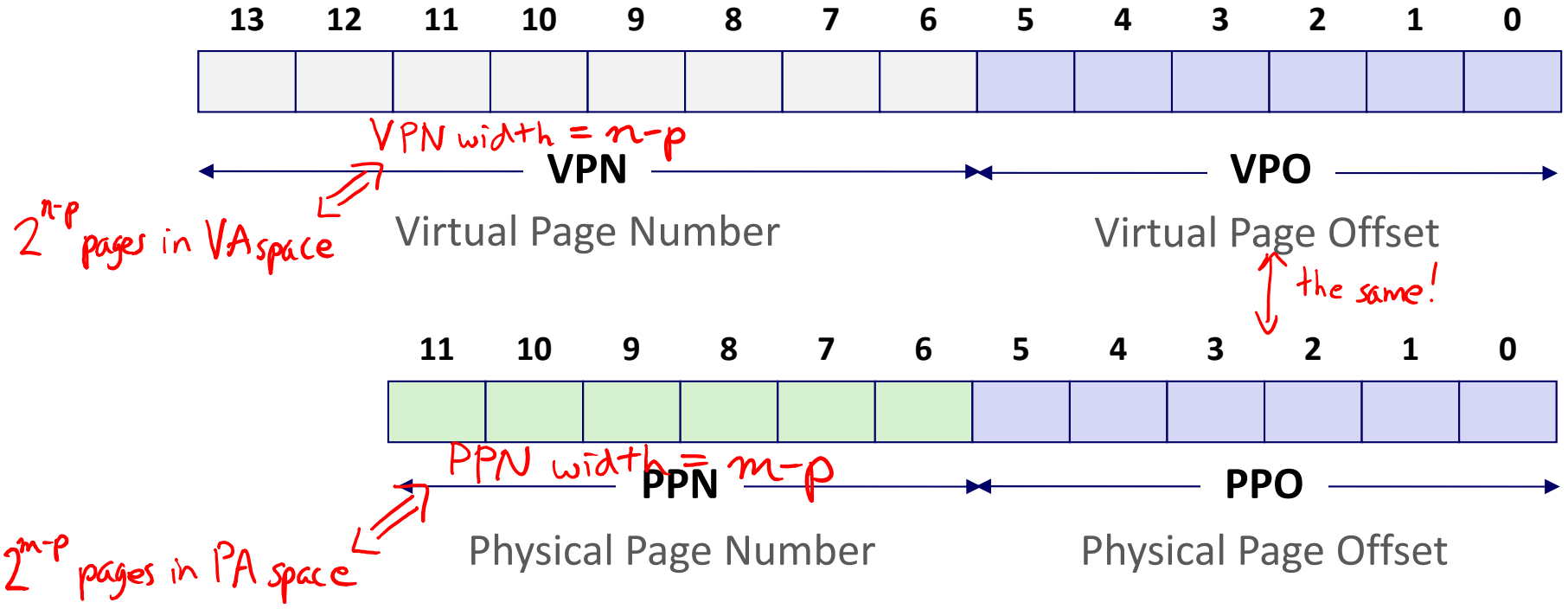
Address Translation

- 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 $n = 14$ bits $\iff$ $N = 16$ KiB VA space
  - 12-bit physical address $m = 12$ bits $\iff$ $M = 4$ KiB PA space
  - Page size = 64 bytes $P = 64$ B $\iff$ $p = 6$ bits



Simple Memory System: Page Table

- Only showing first 16 entries (out of 2^{n-p} = 2^8 = 256) one for every virtual page
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE (D, R, W, X)

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

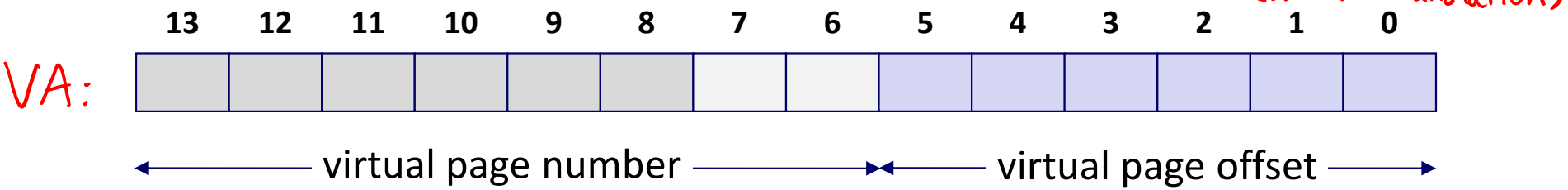
- 16 entries total
- 4-way set associative

16/4 = 4 sets

2 bits

Why does the TLB ignore the page offset?

not part of its job! (address translation)

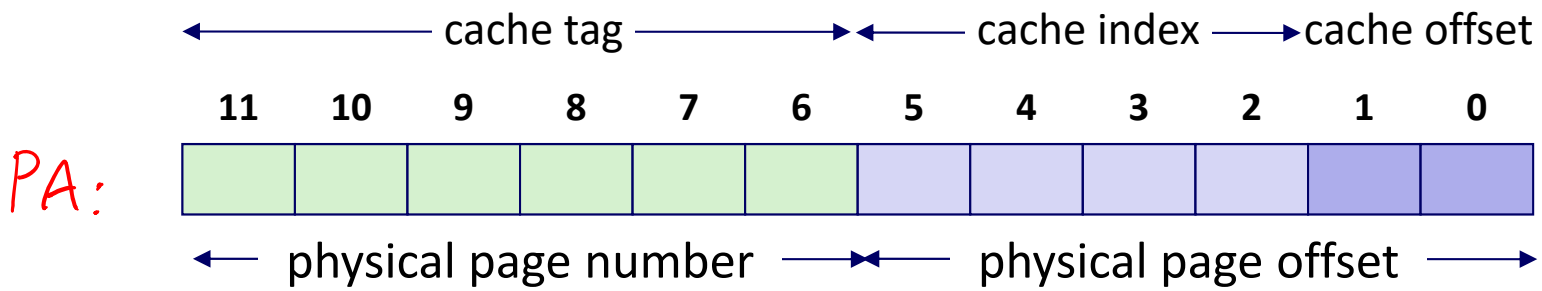


	Way 0			Way 1			Way 2			Way 3		
Set	**Tag**	**PPN**	**Valid**	**Tag**	**PPN**	**Valid**	**Tag**	**PPN**	**Valid**	**Tag**	**PPN**	**Valid**
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

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	VPN	PPN	V
③ 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	② E	–	0✗
7	–	0	F	0D	1

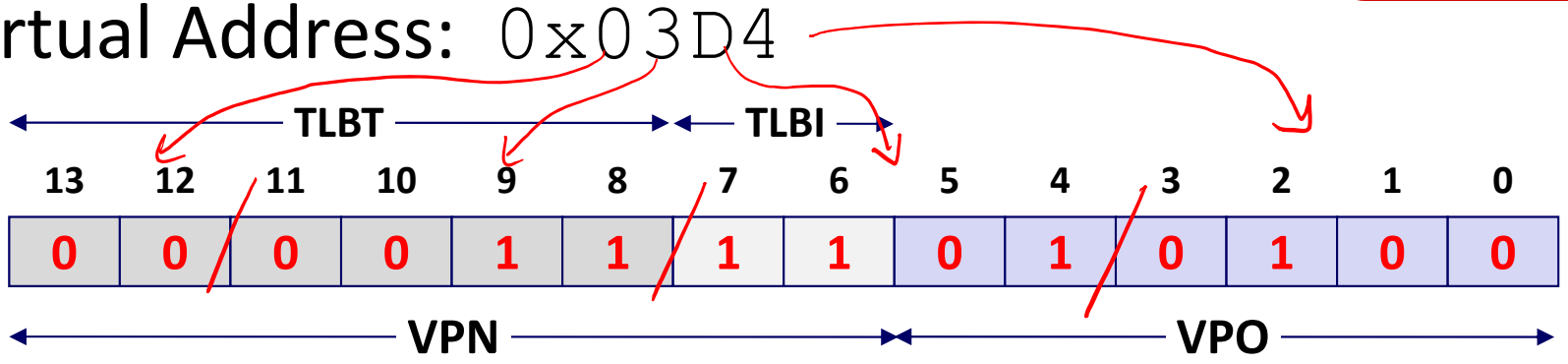
Cache:

Index	Tag	V	B0	B1	B2	B3	Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11	③ 8	24✗	1✓	3A	00	51	89
1	15	0	–	–	–	–	9	2D	0	–	–	–	–
2	1B	1	00	02	04	08	④ A	2D✓	1✓	93	15	DA	3B
3	36	0	–	–	–	–	B	0B	0	–	–	–	–
4	32	1	43	6D	8F	09	C	12	0	–	–	–	–
① 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

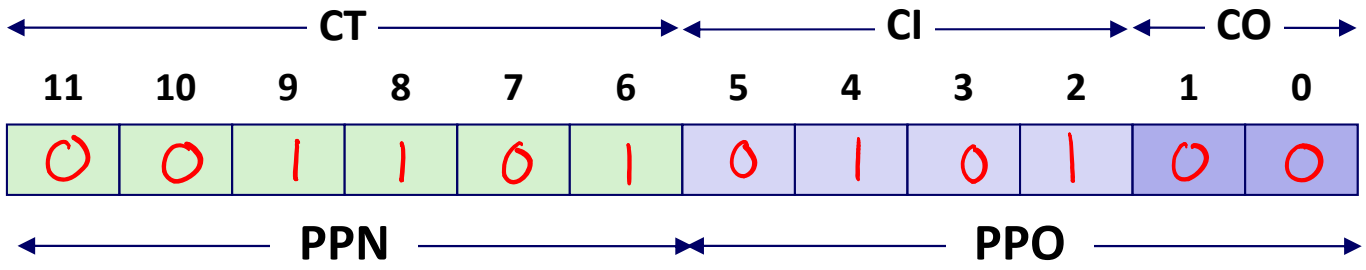
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x03D4



VPN 0xF (check this entry of the page table) TLBT 0x03 (look for this tag within TLB set) TLBI 3 (check this set of the TLB) TLB Hit? Y Page Fault? N PPN 0x0D

- Physical Address:

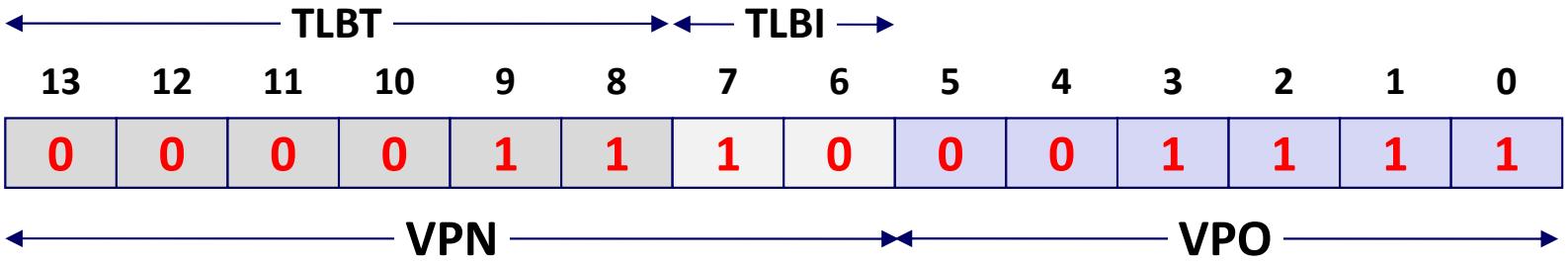


CT 0x0D (look for this tag within cache set) CI 5 (check this set of the cache) CO 0 (which byte of cache block) Cache Hit? Y Data (byte) 0x36

Memory Request Example #2

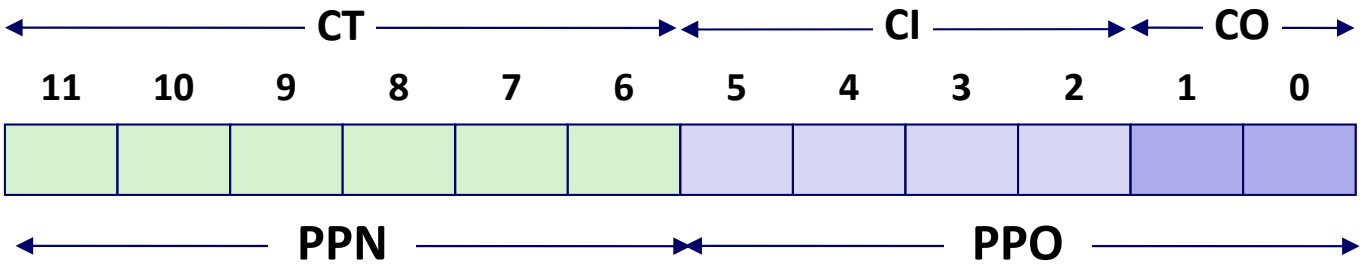
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x038F`



VPN 0x0E TLBT 0x03 TLBI 2 TLB Hit? N Page Fault? Y PPN n/a

- Physical Address:

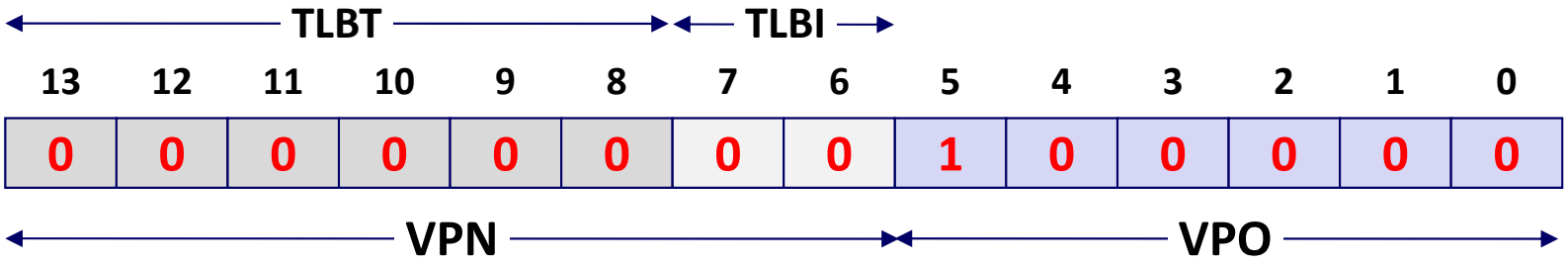


CT _____ CI _____ CO _____ Cache Hit? ___ Data (byte) _____

Memory Request Example #3

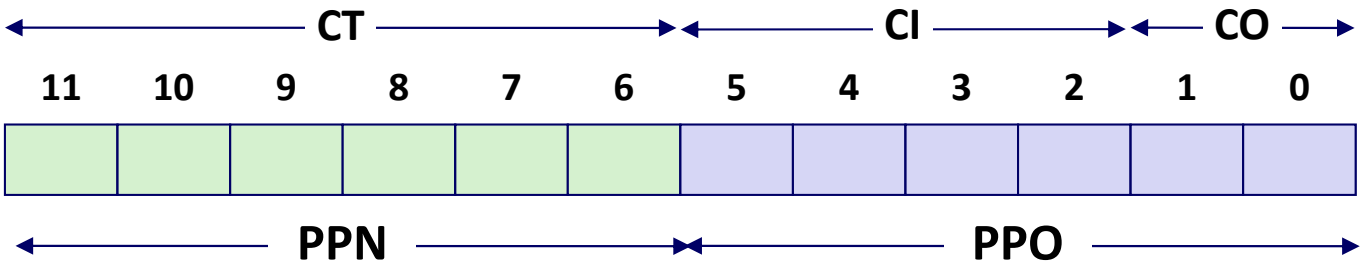
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x0020



VPN 0x00 TLBT 0x00 TLBI 0 TLB Hit? N Page Fault? N PPN 0x28

- Physical Address:

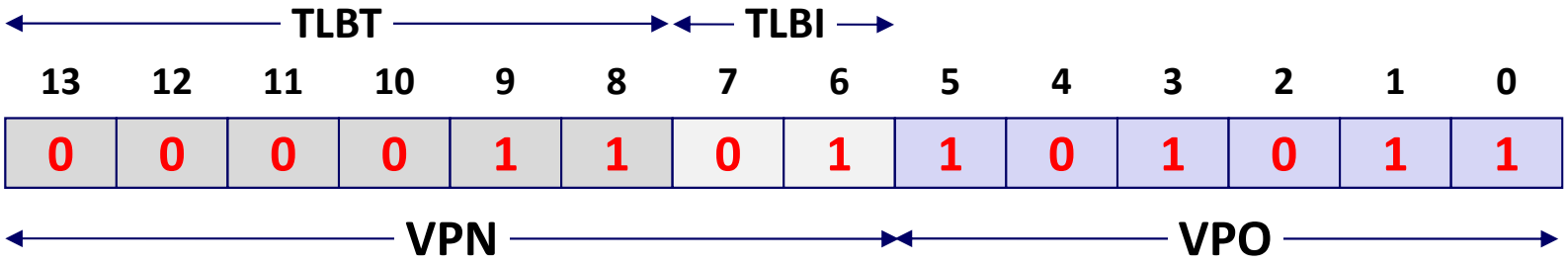


CT 0x28 CI 8 CO 0 Cache Hit? N Data (byte) n/a

Memory Request Example #4

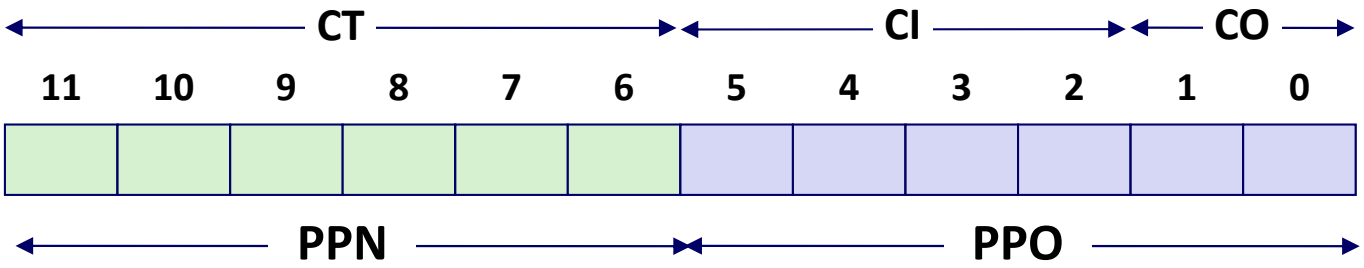
Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: 0x036B



VPN 0x0D TLBT 0x03 TLBI 1 TLB Hit? Y Page Fault? N PPN 0x2D

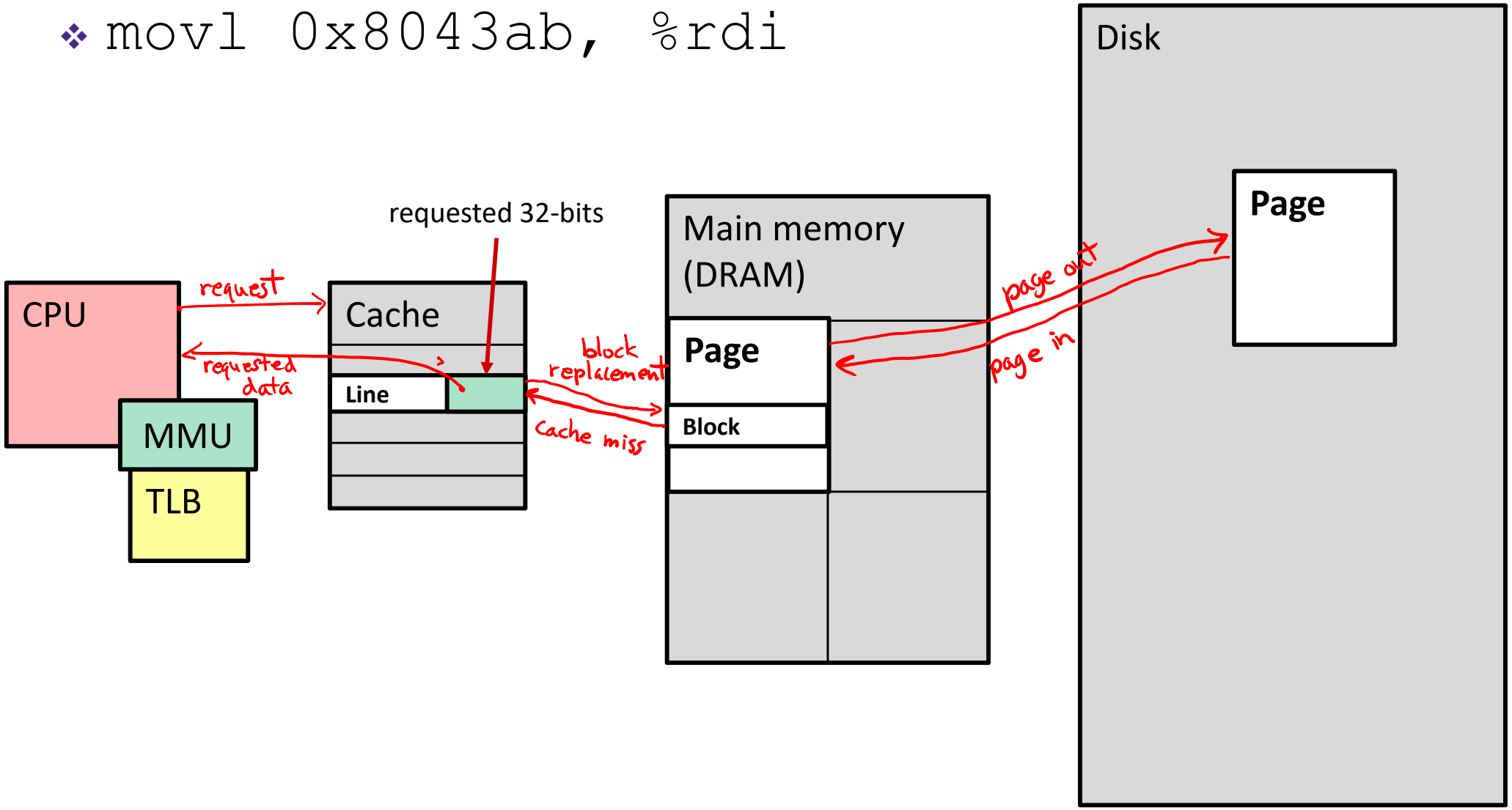
- Physical Address:



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

Memory Overview (data flow)

- `movl 0x8043ab, %rdi`



Page Table Reality

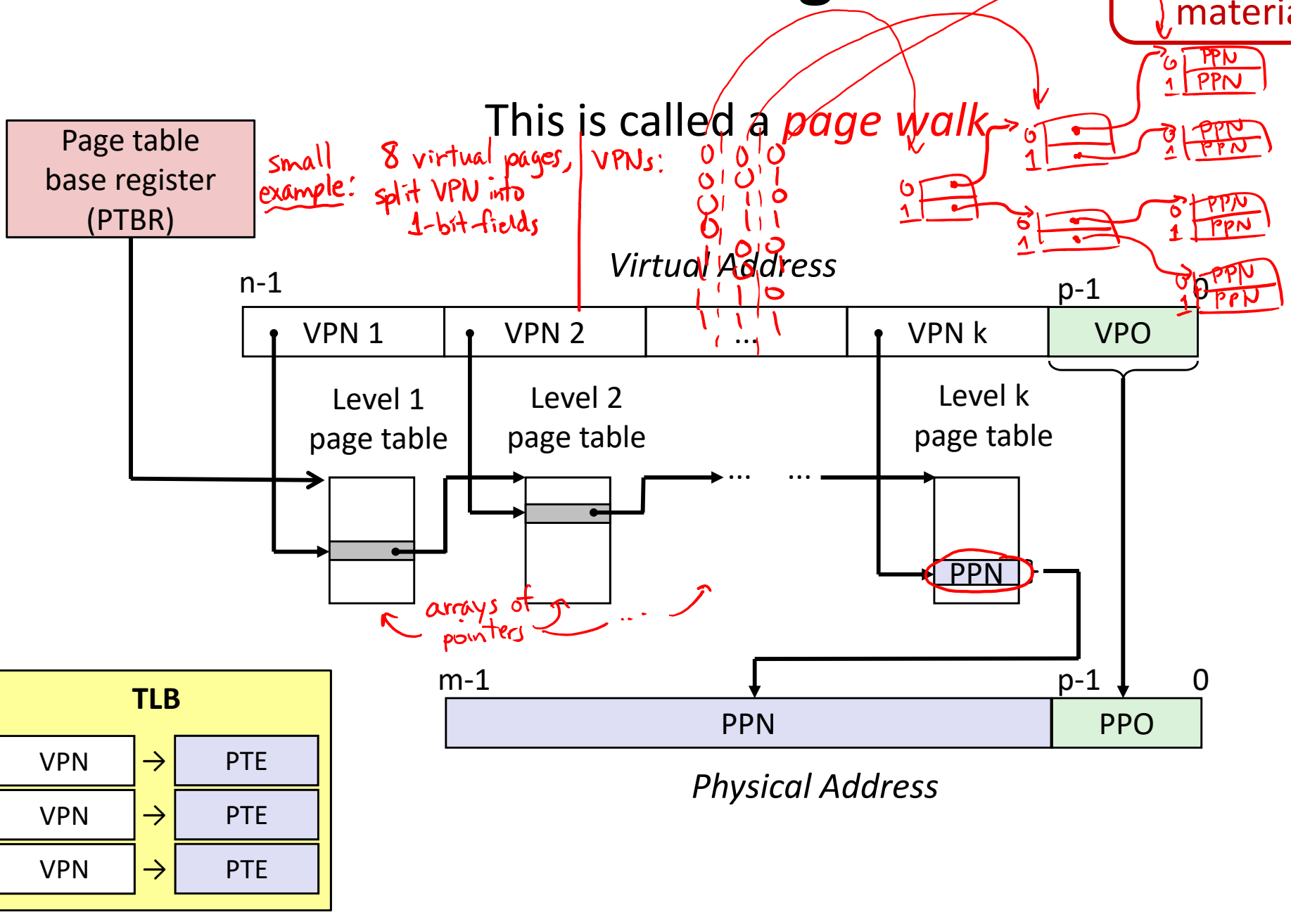
This is extra (non-testable) material

- 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
  - How many page table entries is that?
    1 PTE for every virtual page
    $2^{n-p} = 2^{51}$ PTEs
  - About how long is each PTE?
    PPN width + management bits = 20+5 = 25 bits ≈ 3 bytes
    m-p (V,D,R,W,X)
  - $\approx 2^{52} + 2^{51}$ bytes per page table!
  - **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



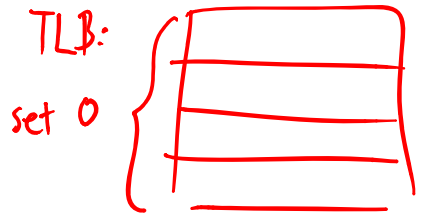
Multi-level Page Tables

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} Entries in a page table (2^{n-p} ← # of virtual pages)

20 Minimum bit-width of PTBR ← physical address of PT (m)

17 TLBT bits (VPN → TLBT / TLBI; here TLBI = 0)

2^5 Max # of valid entries in a page table (2^{m-p}) ← # of pages in physical memory

Practice VM Question

starting address of matrix is at page offset of 0

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

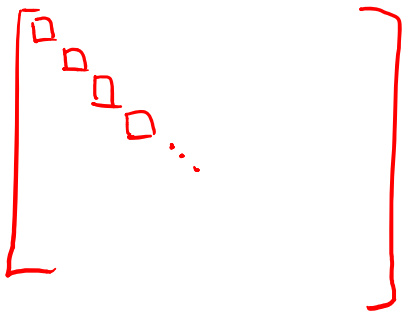
```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

updating diagonal entries

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

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

stride is always 2049 ints = 2049*4 bytes



Practice VM Question

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

- 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;
```

(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?

3/4 = 75% TLB Hit Rate

0% Page Table Hit Rate

access pattern: single write to index
never revisit indices (always increasing)
we access every row of matrix exactly once

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

within each page: M HHH

only access PT on TLB Miss
because mat[] on disk, each first access to page causes page fault.

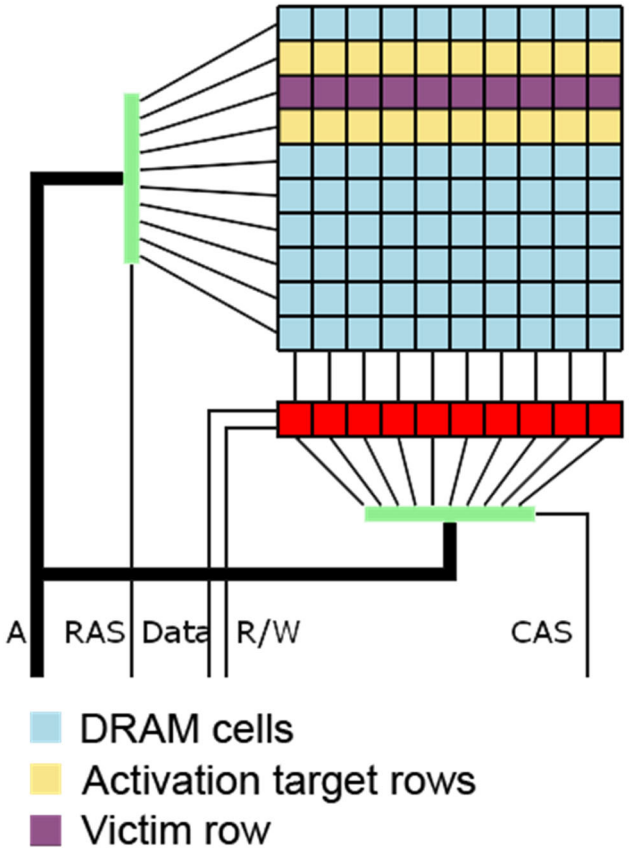
BONUS SLIDES

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

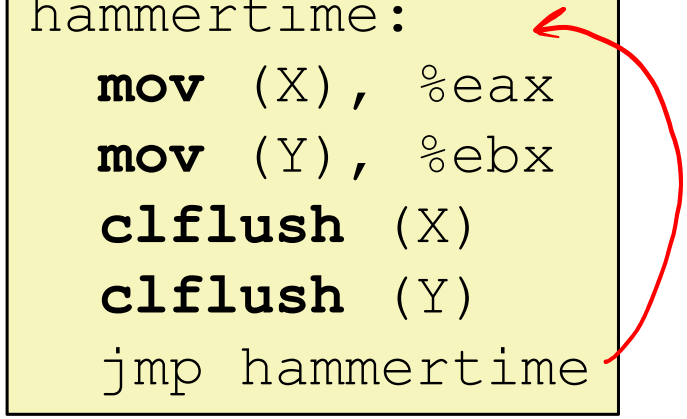


By Dsimic (modified), CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38868341

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

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



- 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?

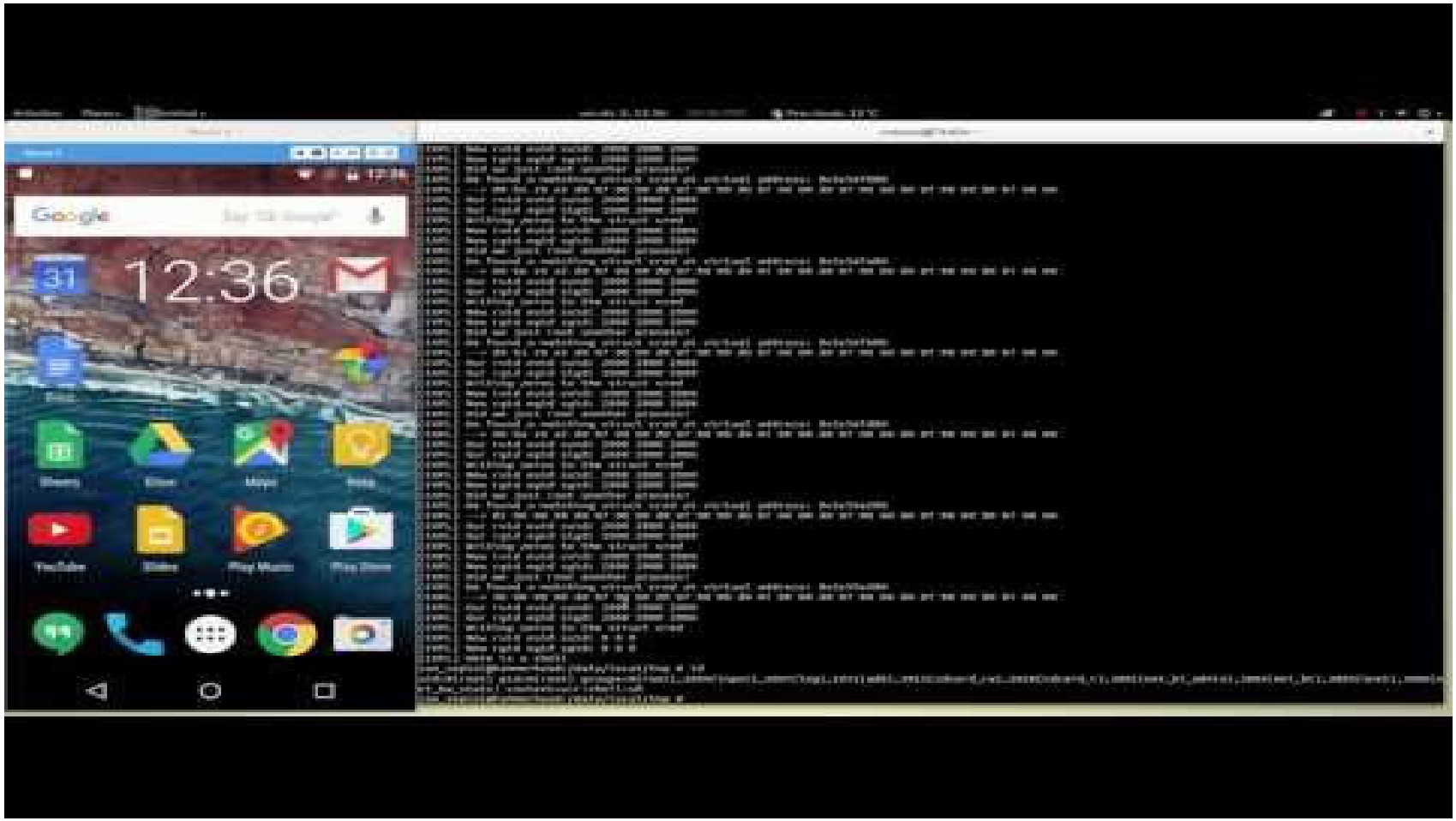
- Doesn't seem so bad – random bit flip in a row of physical memory
  - Vulnerability affected by system setup and physical condition of memory cells
- **Improvements:**
  - Double-sided row hammering increases speed & chance
  - Do system identification first (e.g. Lab 4)
    - Use timing to infer memory 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