

Virtual Memory III

CSE 351 Autumn 2020

Instructor:

Justin Hsia

Teaching Assistants:

Aman Mohammed

Ami Oka

Callum Walker

Cosmo Wang

Hang Do

Jim Limprasert

Joy Dang

Julia Wang

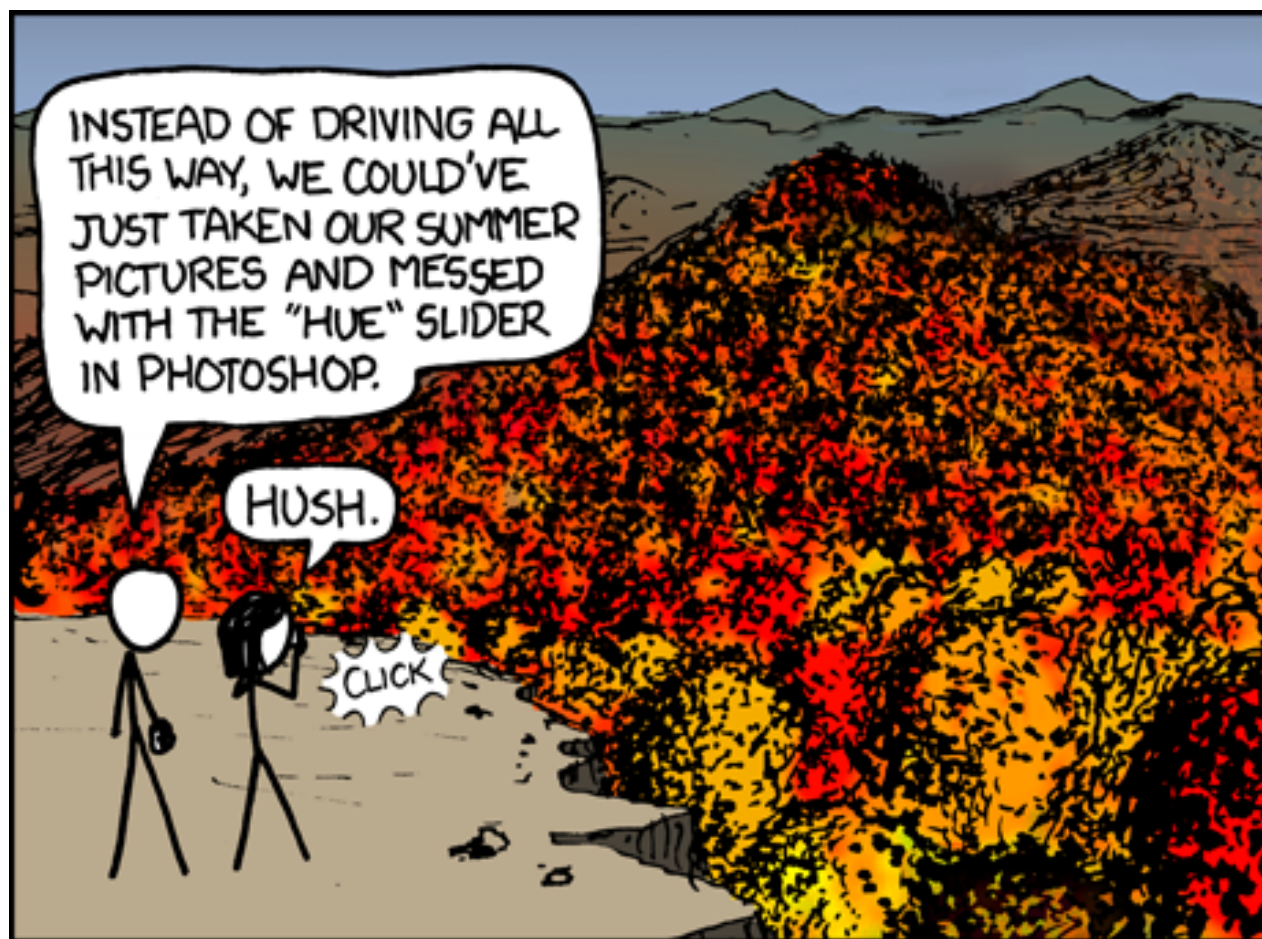
Kaelin Laundry

Kyrie Dowling

Mariam Mayanja

Shawn Stanley

Yan Zhe Ong



<https://xkcd.com/648/>

Administrivia

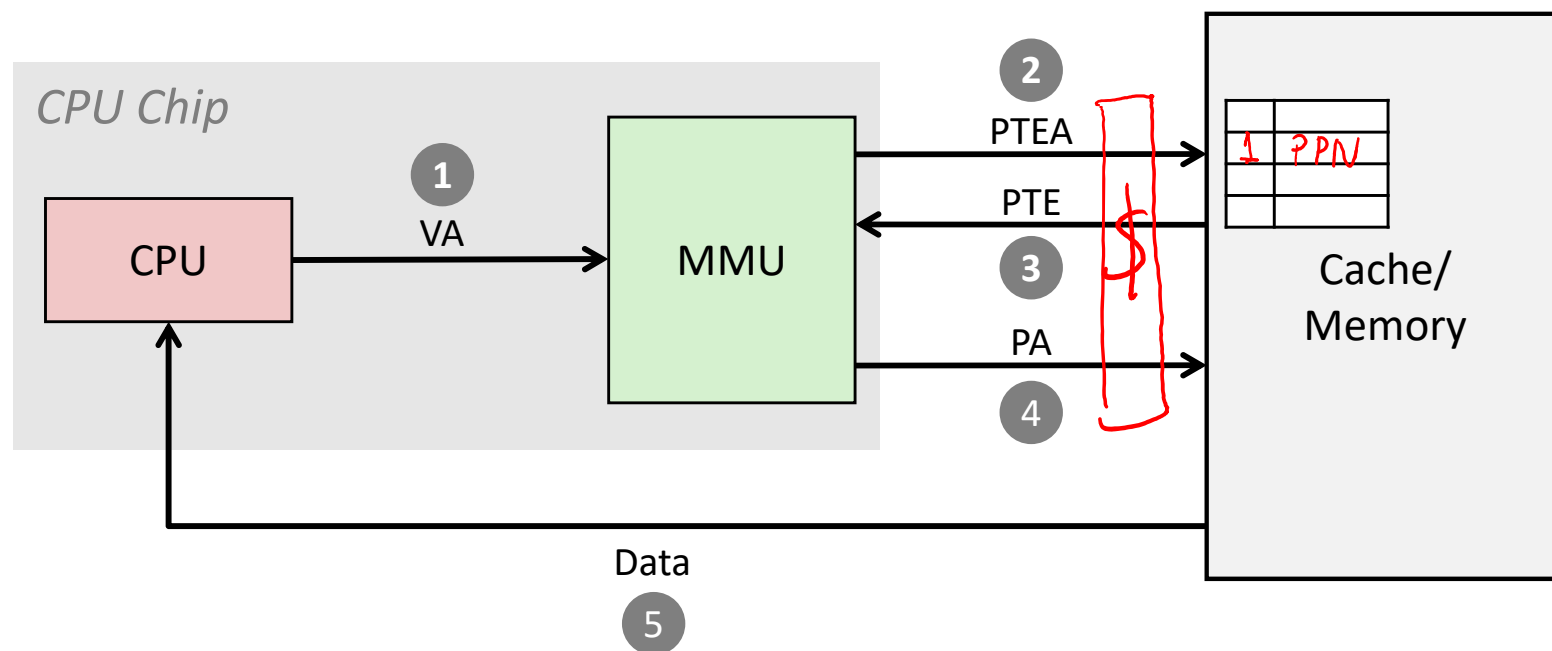
- ❖ Lab 4 due Monday (11/30)
- ❖ hw22 due *Wednesday* (12/2)
- ❖ “Virtual Section” on Virtual Memory
 - Worksheet and solutions released on Wednesday or Thursday
 - Videos will be released of material review and problem solutions
- ❖ Midterm retake Dec. 3-4 (Thu & Fri)
 - Opt-in survey on Canvas open until Dec. 2 @ 1 pm

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 (page does live in physical mem)

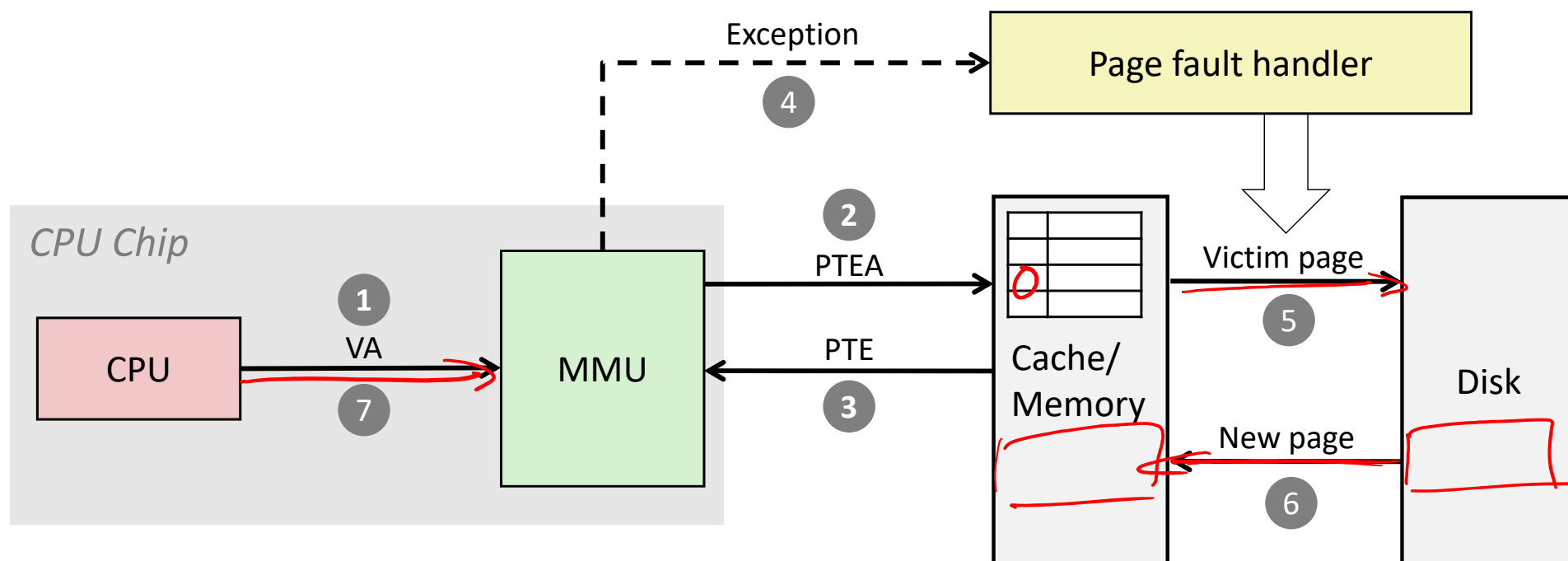


- 1) Processor sends *virtual* address to MMU (*memory management unit*)
- 2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)
- 4) MMU sends *physical* address to cache/memory requesting data
- 5) Cache/memory sends data to processor

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


(page is NOT in physical mem)



- 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

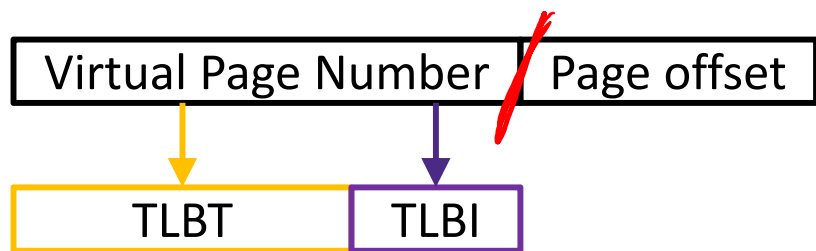
Hmm... Translation Sounds Slow

- ❖ 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

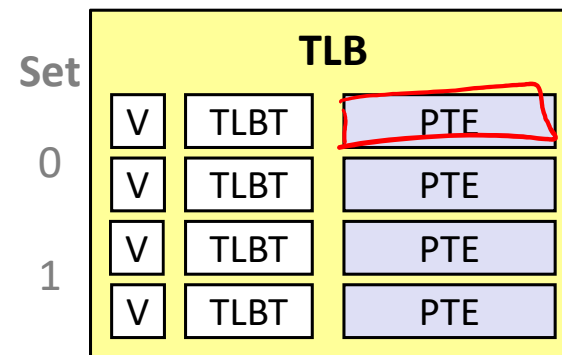
- ❖ *What can we do to make this faster?*
 - **Solution:** add another cache! 

Speeding up Translation with a TLB

- ❖ *PTE* *cache*
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

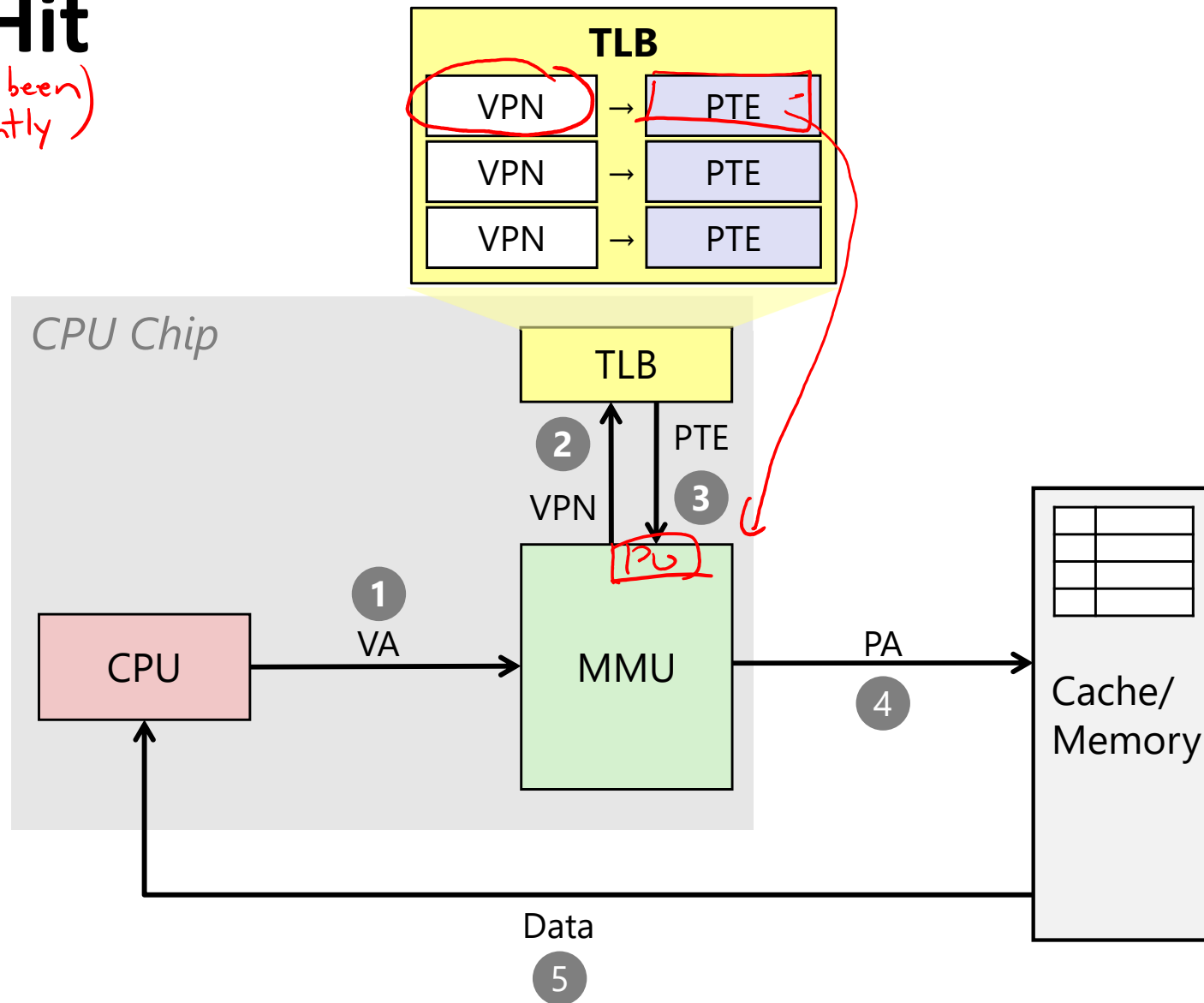


VPN % (# of sets in the TLB)



TLB Hit

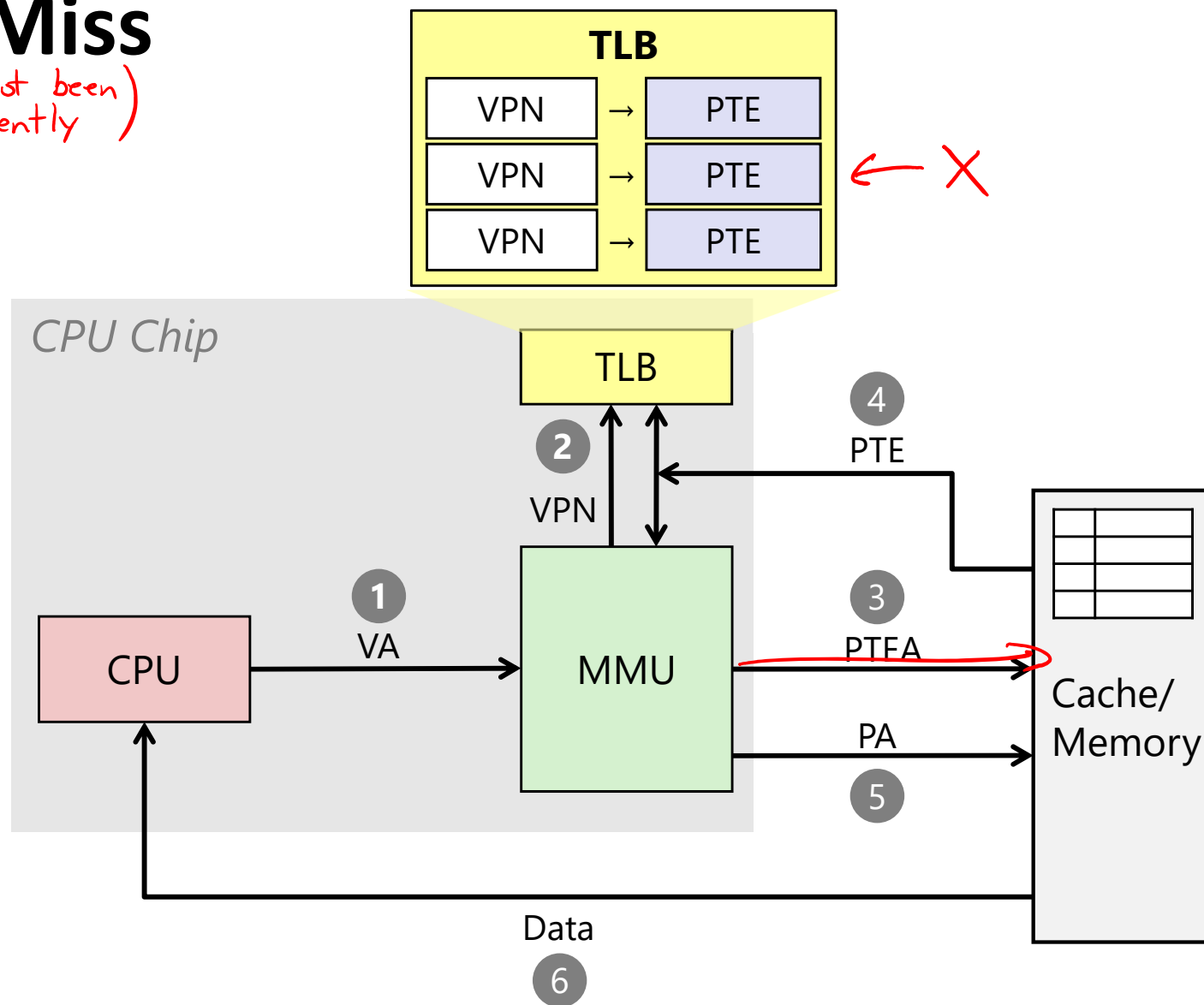
(page has been used recently)



❖ A TLB hit eliminates a memory access!

TLB Miss

(page has not been used recently)



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

Fetching Data on a Memory Read

1) Check TLB *(translate VA → PA)*

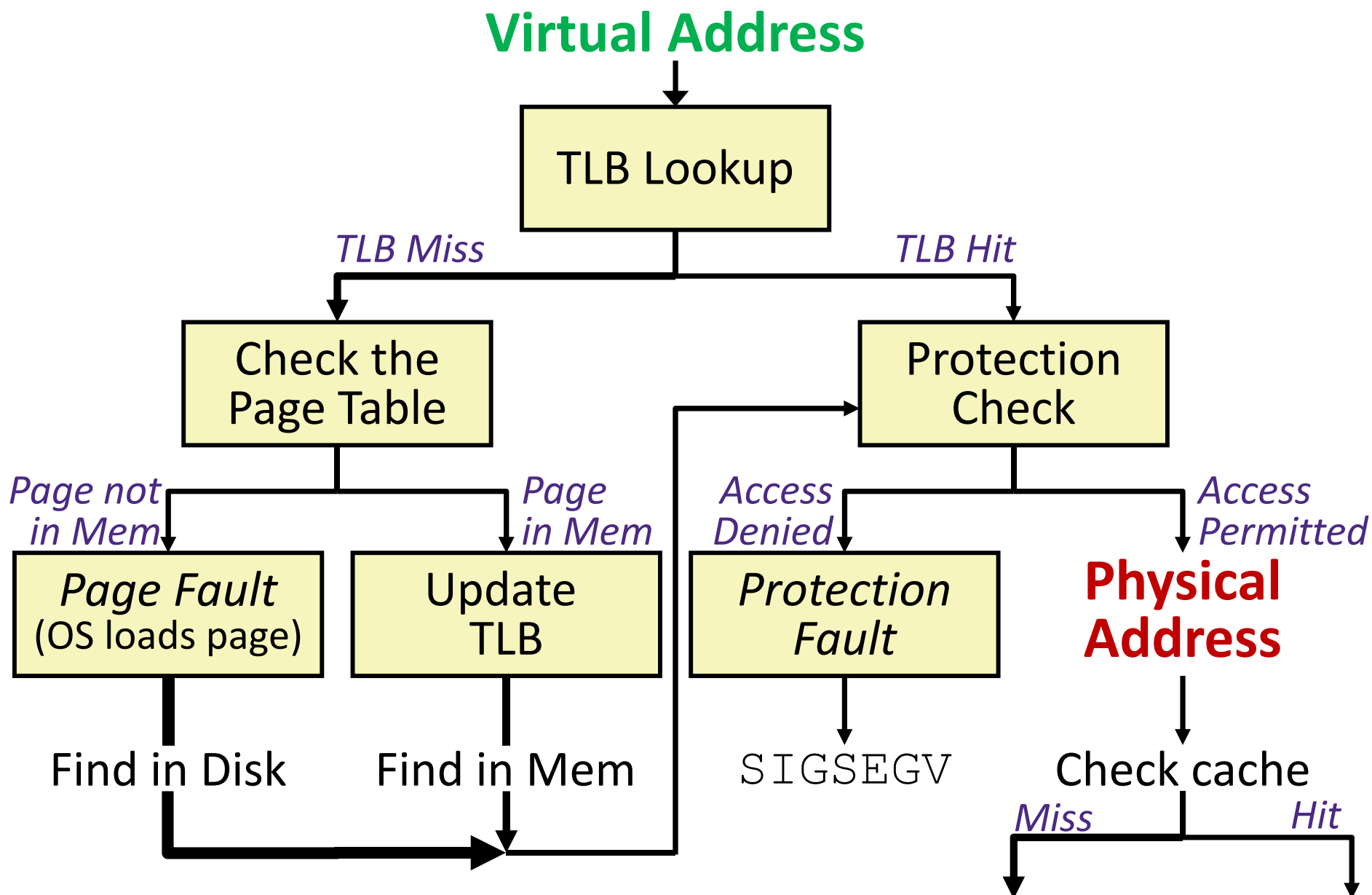
- Input: VPN, Output: PPN
- **TLB Hit:** Fetch translation, return PPN
- **TLB Miss:** Check page table (in memory)
 - **Page Table Hit:** Load page table entry into TLB
 - **Page Fault:** Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB

these serve different purposes!

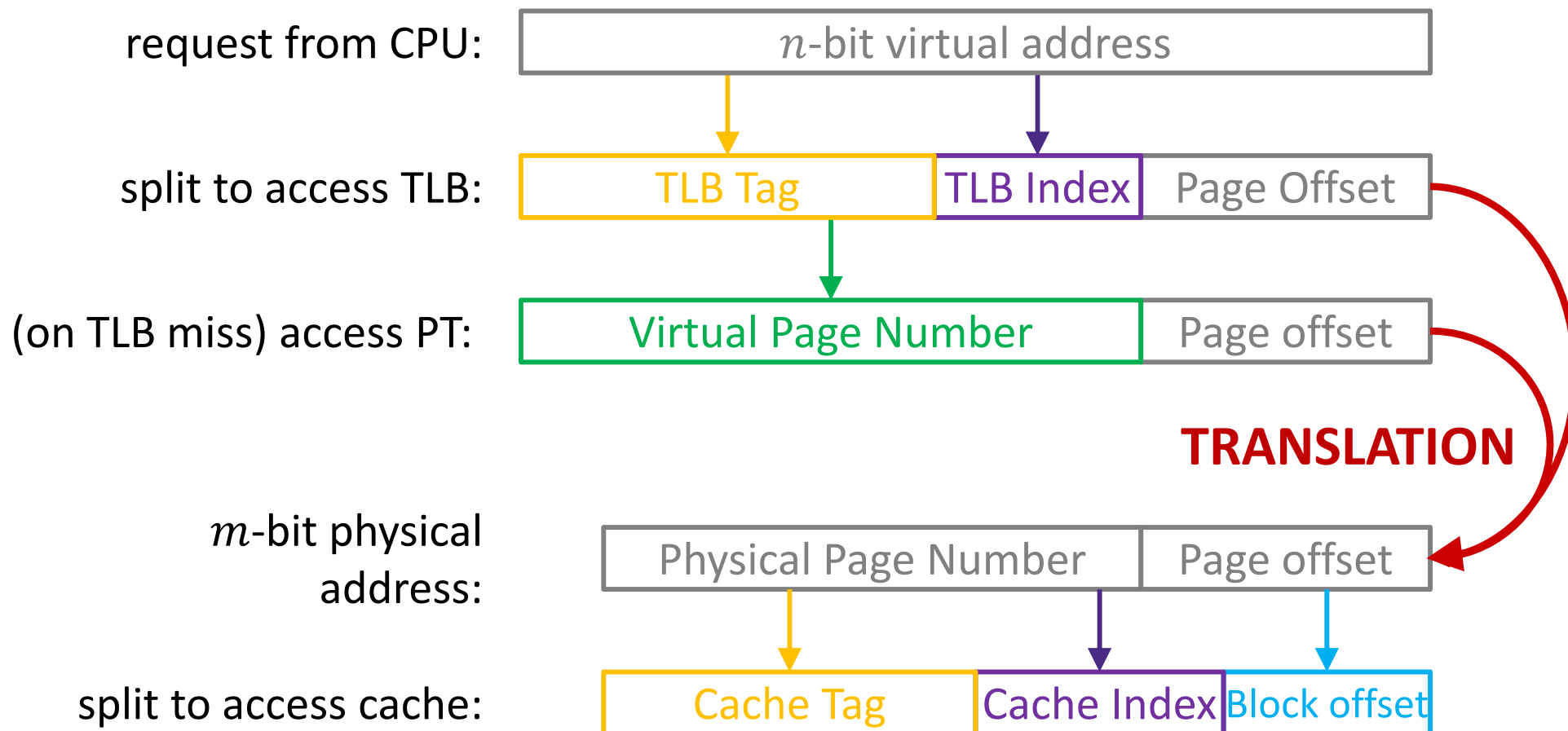
2) Check cache *(fetch requested data)*

- 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



Address Manipulation



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

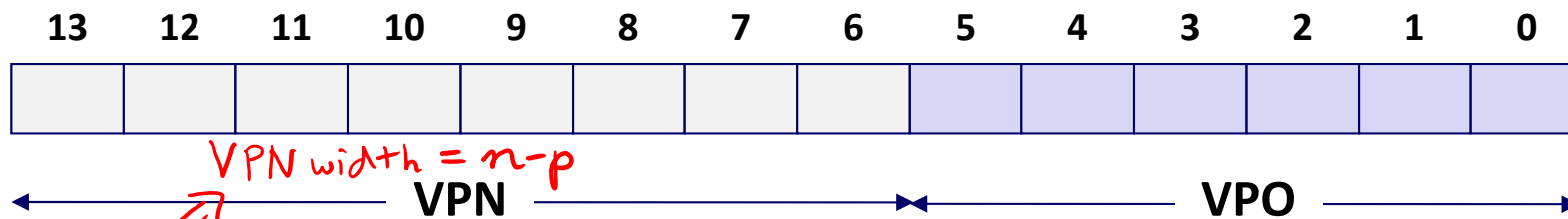
❖ Components of the physical address (PA)

- **PPO** Physical page offset (same as VPO)
- **PPN** Physical page number

Simple Memory System Example (small)

❖ Addressing

- 14-bit virtual addresses $n = 14$ 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

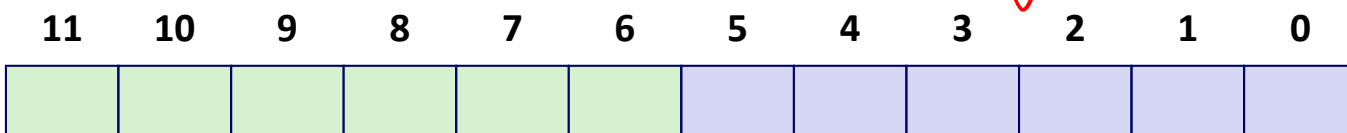


2^{n-p} pages in VA space

Virtual Page Number

Virtual Page Offset

the same!



2^{m-p} pages in PA space

Physical Page Number

Physical Page Offset

Simple Memory System: Page Table

- ❖ Only showing first 16 entries (out of 2^{n-p} 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	0x13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1
⋮	⋮	⋮
⋮	⋮	⋮
⋮	⋮	⋮

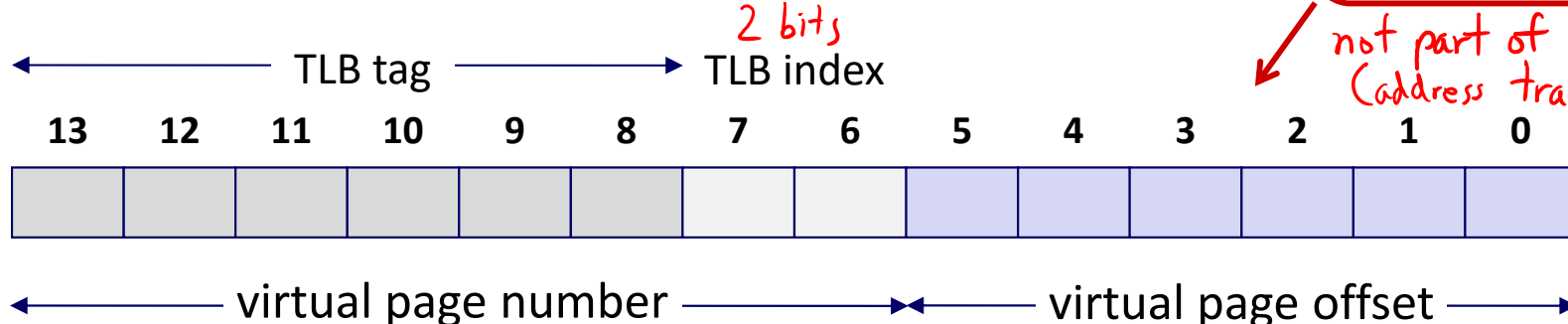
Simple Memory System: TLB

- ❖ 16 entries total
- ❖ 4-way set associative

$16/4 = 4$ sets

Why does the TLB ignore the page offset?

not part of its job! (address translation)

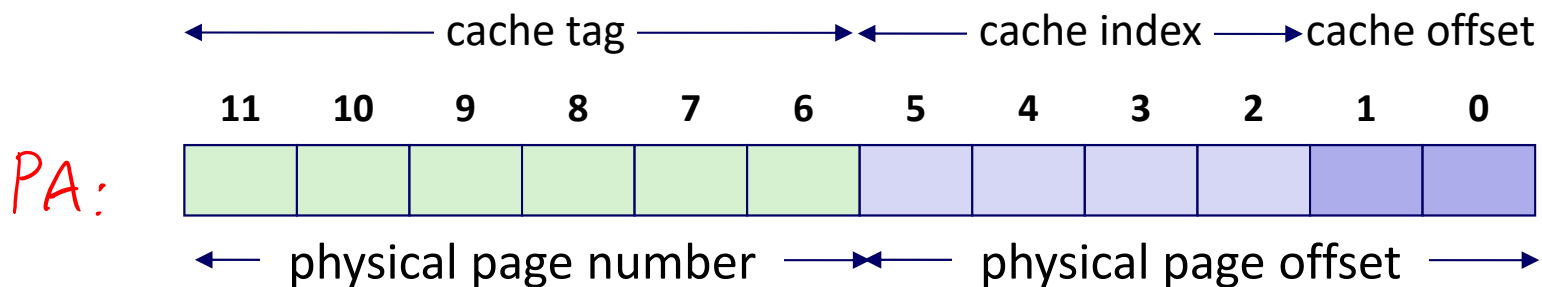


Set	Way 0			Way 1			Way 2			Way 3		
	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

Simple Memory System: Cache

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

- ❖ Direct-mapped with $K = 4 \text{ B}$, $C/K = 16$
- ❖ 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	<u>00</u>	-	0 X	07	02	1
④ 1	<u>03</u>	<u>2D</u>	1✓	02	-	0	04	-	0	0A	-	0
② 2	02	-	0	08	-	0	06	-	0	<u>03</u>	-	0 X
① 3	07	-	0	<u>03</u>	<u>0D</u>	1✓	0A	34	1	02	-	0

Page table (partial):

VPN	PPN	V	VPN	PPN	V
③ 0	<u>28</u>	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 X
7	-	0	F	0D	1

Cache:

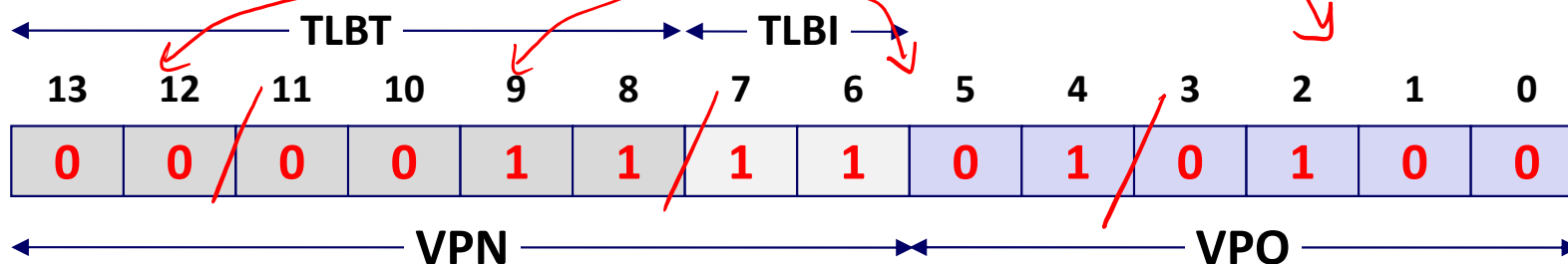
Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
① 5	0D ✓	1 ✓	<u>36</u>	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

Index	Tag	V	B0	B1	B2	B3
③ 8	24 X	1✓	3A	00	51	89
9	2D	0	-	-	-	-
④ A	2D ✓	1 ✓	93	15	DA	<u>3B</u>
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

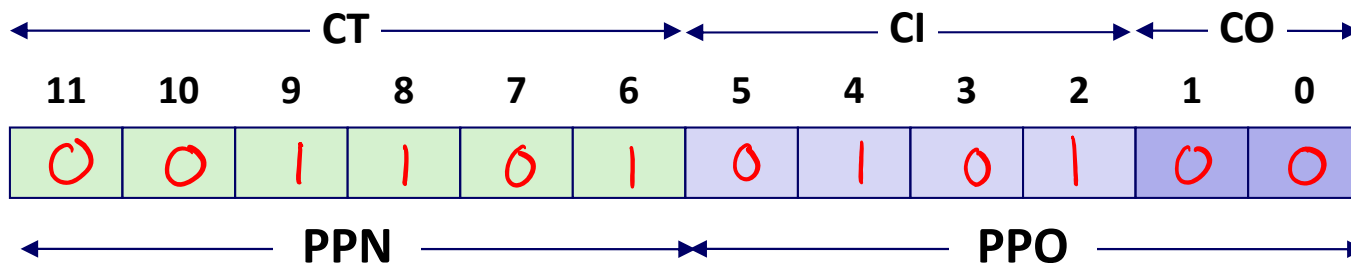
Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x03D4



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

❖ Physical Address:

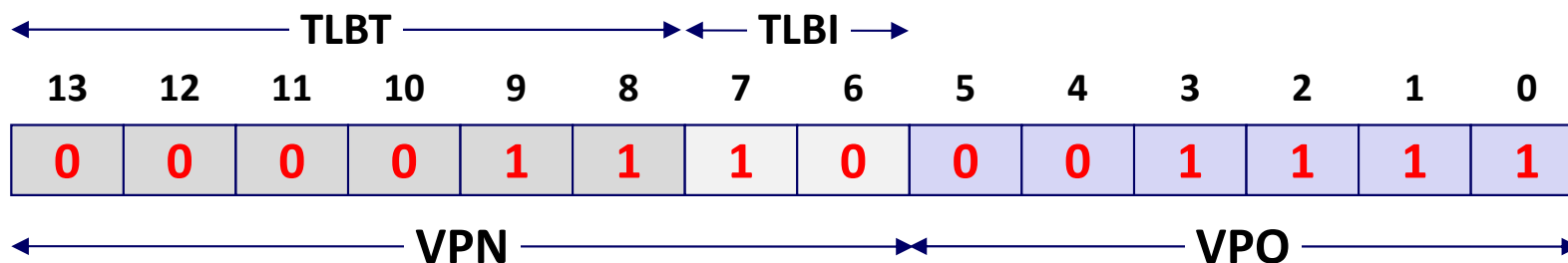


CT 0x0D CI 5 CO 0 Cache Hit? Y Data (byte) 0x36
look for this tag within cache set *check this set of the cache* *which byte of cache block*

Memory Request Example #2

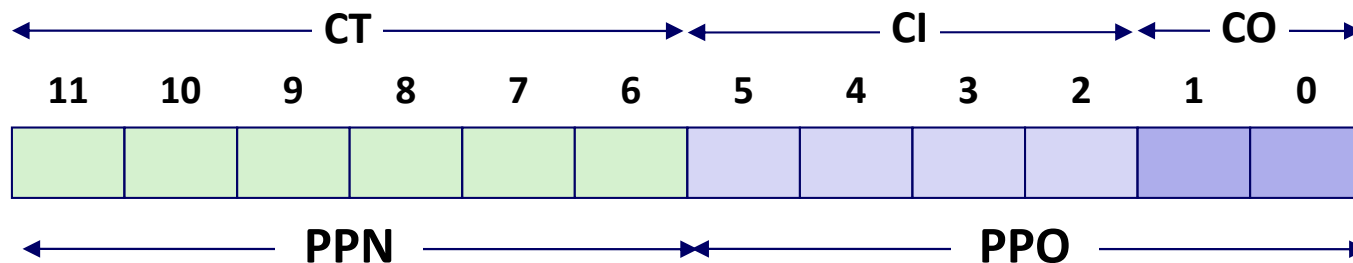
Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: $0x038F$



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

❖ Physical Address:

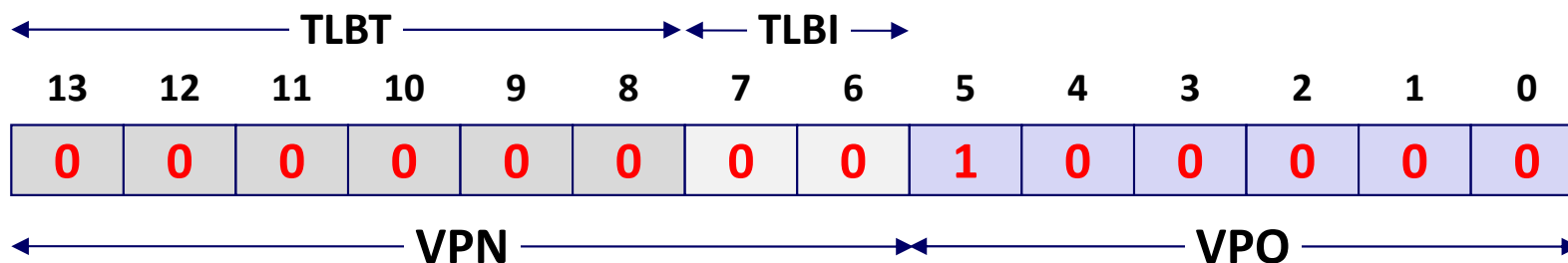


CT _____ CI _____ CO _____ Cache Hit? _____ Data (byte) _____

Memory Request Example #3

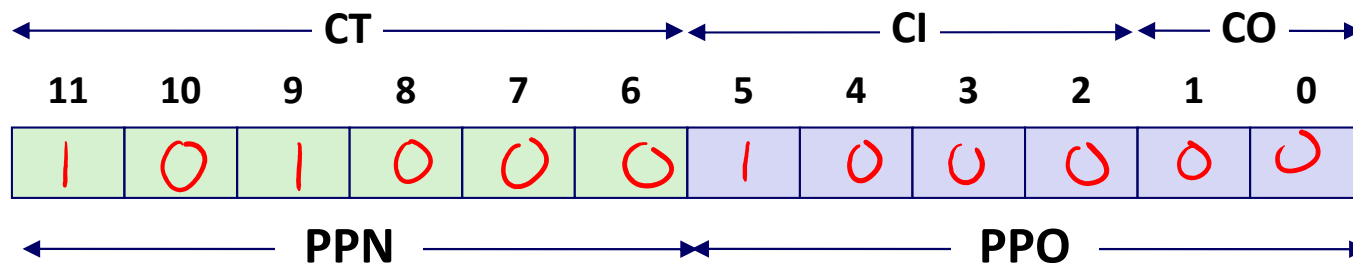
Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x0020



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

❖ Physical Address:

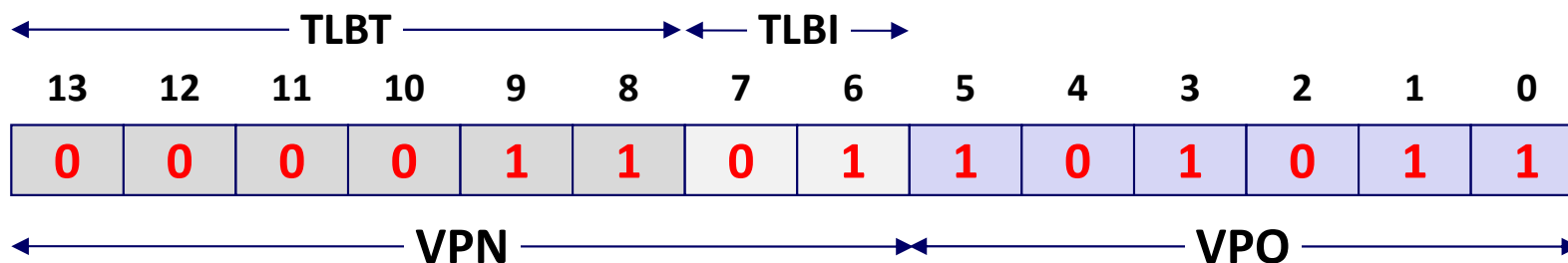


CT 0x28 CI 8 CO 0 Cache Hit? N Data (byte) n/a

Memory Request Example #4

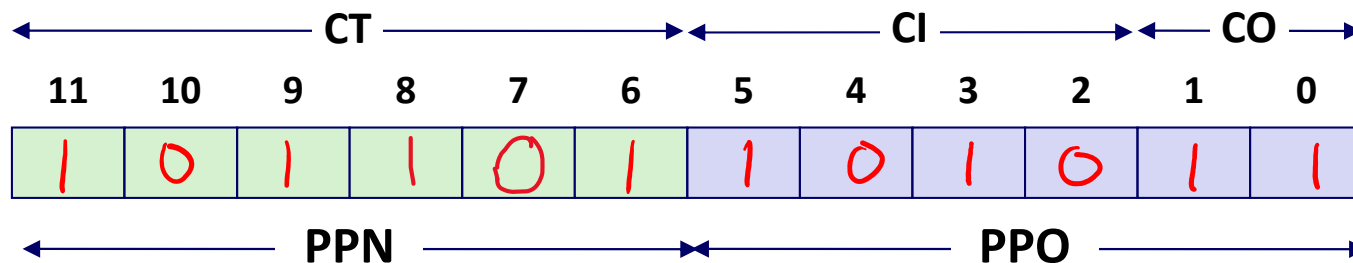
Note: It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x036B



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

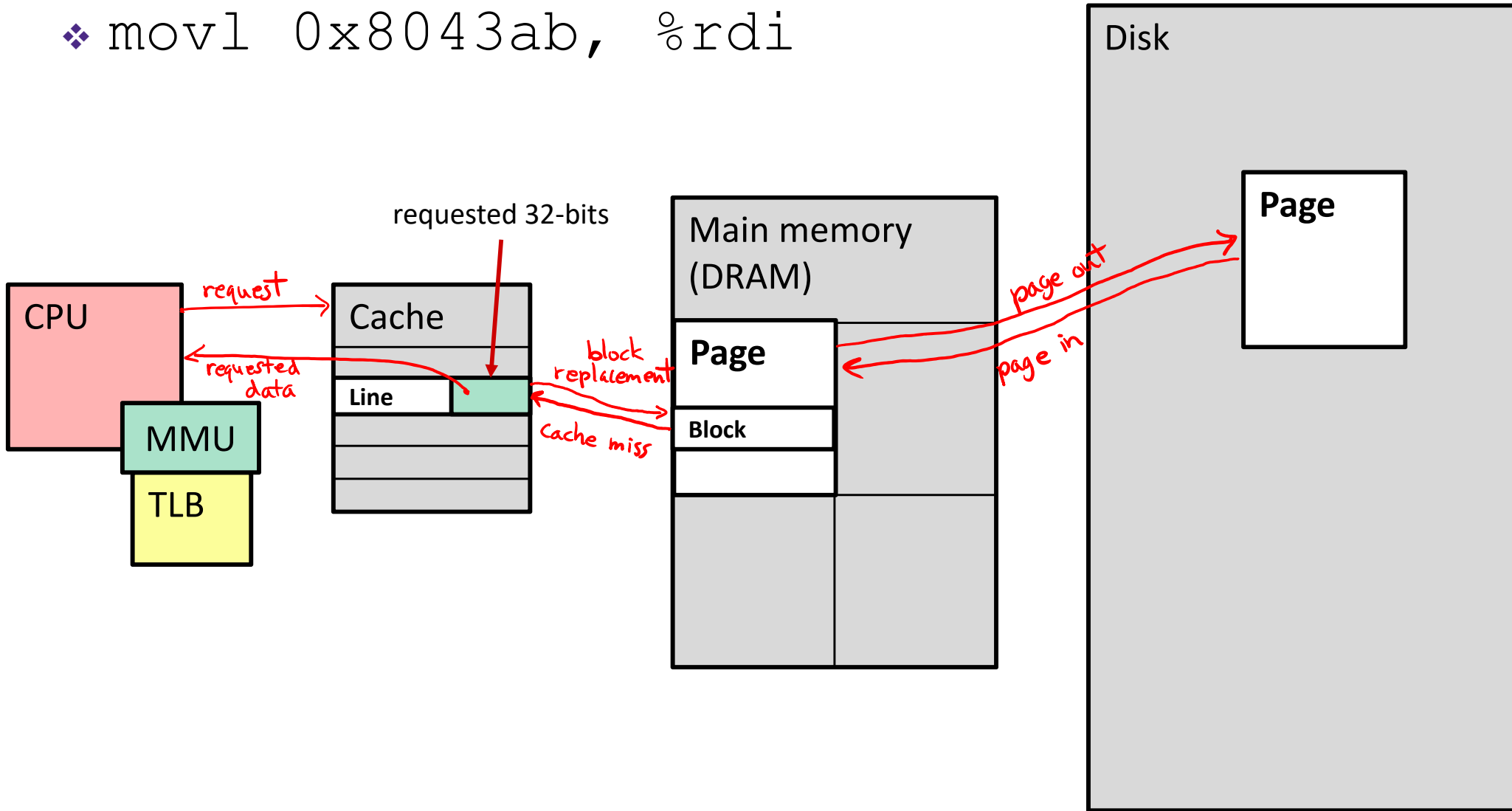
❖ Physical Address:



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:

n = 64 bits, p = 13 bits, m = 33 bits

■ Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory

■ How many page table entries is that?

1 PTE for every virtual page

$2^{n-p} = 2^{51}$ PTEs

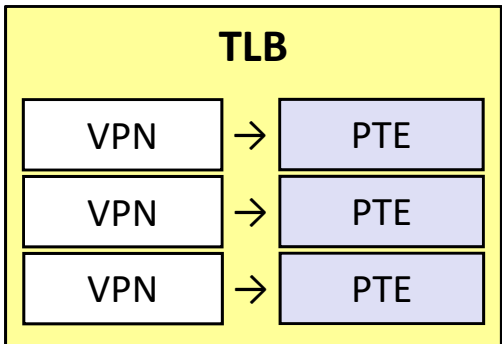
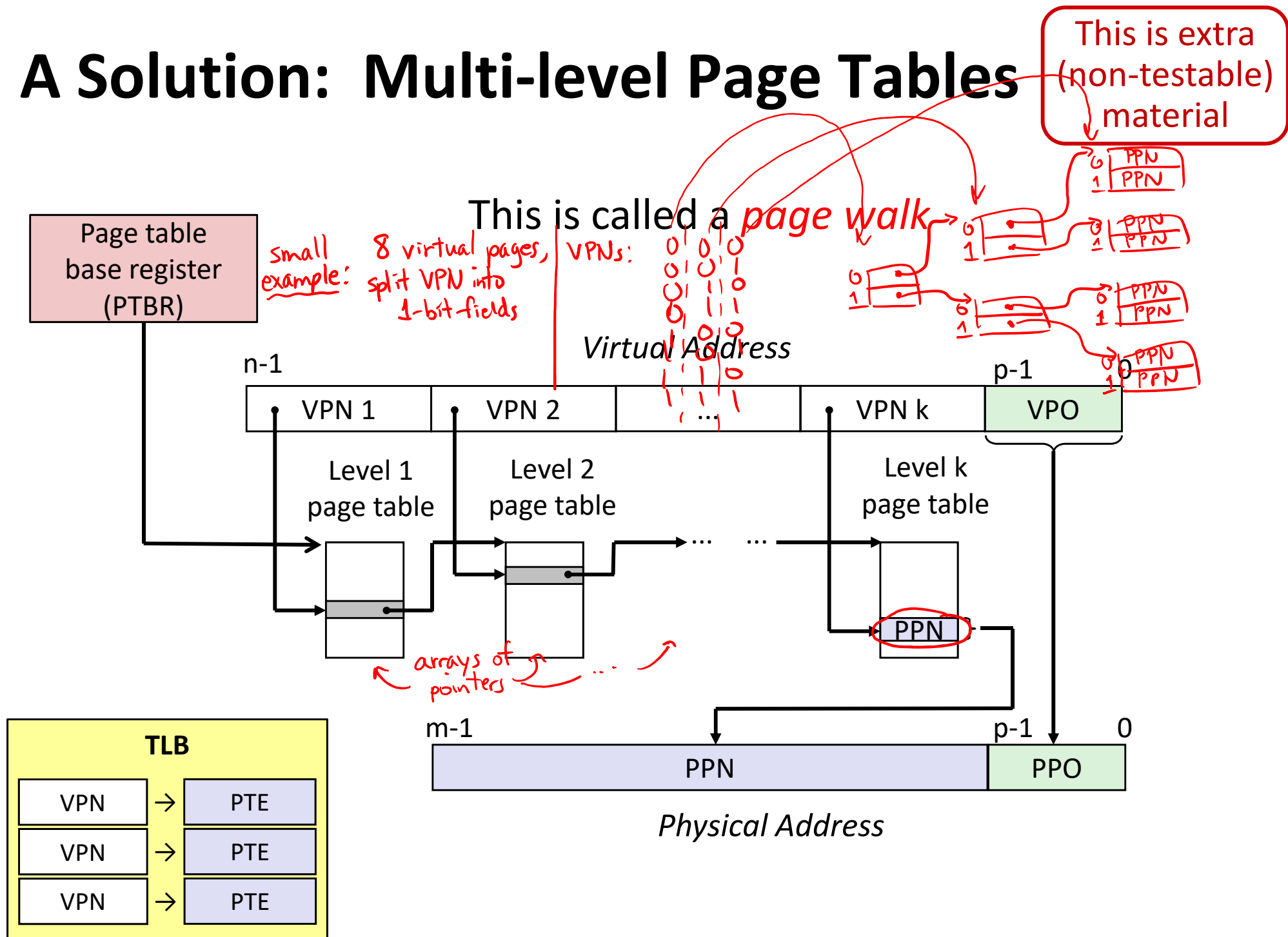
■ About how long is each PTE?

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



Multi-level Page Tables

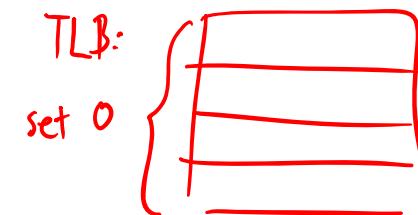
This is extra
(non-testable)
material

- ❖ 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
 $2^{n-p} \leftarrow \#$ of virtual pages

20 Minimum bit-width of PTBR
 \leftarrow physical address of PT
 m

17 TLBT bits
 $VPN \rightarrow TLBT / TLBI$
 here $TLBI = 0$

2^5 Max # of valid entries in a page table
 $\leftarrow \#$ of pages in physical memory
 2^{m-p}

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:

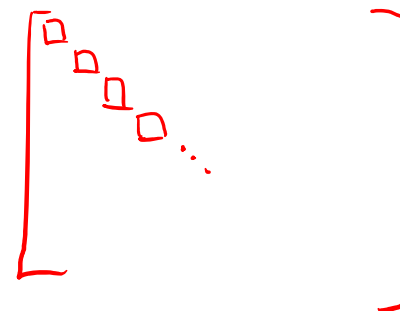
```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
    mat[i*(MAT_SIZE+1)] = i;
```

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

updating diagonal entries

<u>i</u>	<u>array index accessed</u>
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 = 2112048 ints = 213 B
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?

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

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

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

within each page: M H H H

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

Memory System Summary

- ❖ Memory Caches (L1/L2/L3)
 - Purely a speed-up technique
 - Behavior invisible to application programmer and (mostly) OS
 - Implemented totally in hardware
- ❖ Virtual Memory
 - Supports many OS-related functions
 - Process creation, task switching, protection
 - Operating System (software)
 - Allocates/shares physical memory among processes
 - Maintains high-level tables tracking memory type, source, sharing
 - Handles exceptions, fills in hardware-defined mapping tables
 - Hardware
 - Translates virtual addresses via mapping tables, enforcing permissions
 - Accelerates mapping via translation cache (TLB)