Virtual Memory III

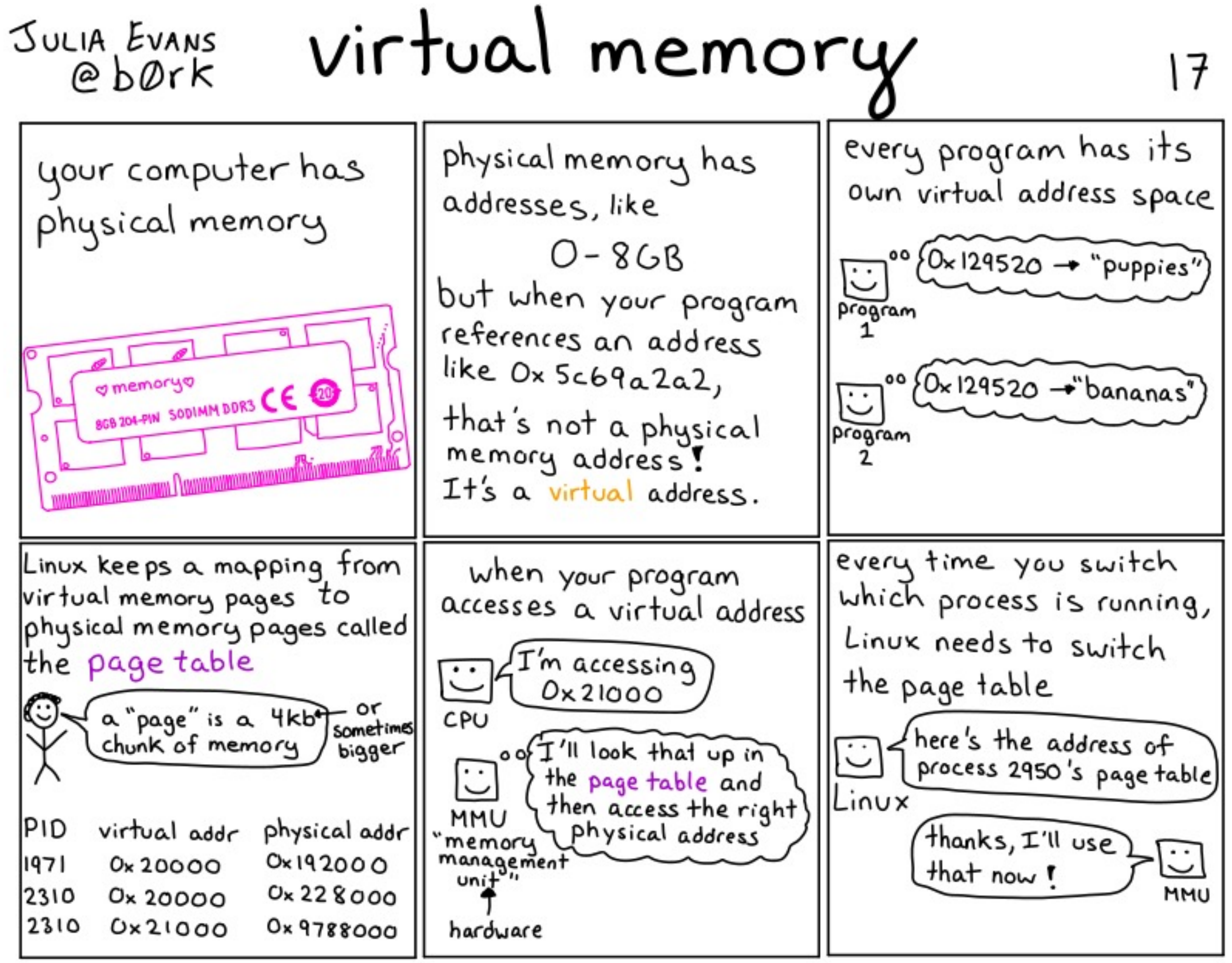
CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker	Maggie Jiang
Adithi Raghavan	Malak Zaki
Aman Mohammed	Naama Amiel
Brenden Page	Nikolas McNamee
Celestine Buendia	Shananda Dokka
Chloe Fong	Stephen Ying
Claire Wang	Will Robertson
Hamsa Shankar	


JULIA EVANS
@b0rk
virtual memory
17
your computer has physical memory
♡ memory ♡
8GB 204-PIN SODIMM DDR3
physical memory has addresses, like
0-8GB
but when your program references an address like 0x5c69a2a2,
that's not a physical memory address! It's a virtual address.
every program has its own virtual address space
0x129520 → "puppies"
program 1
0x129520 → "bananas"
program 2
Linux keeps a mapping from virtual memory pages to physical memory pages called the page table
a "page" is a 4kb chunk of memory
or sometimes bigger
when your program accesses a virtual address
I'm accessing 0x21000
CPU
I'll look that up in the page table and then access the right physical address
MMU "memory management unit"
hardware
every time you switch which process is running, Linux needs to switch the page table
here's the address of process 2950's page table
Linux
thanks, I'll use that now!
MMU

PID	virtual addr	physical addr
1971	0x20000	0x192000
2310	0x20000	0x228000
2310	0x21000	0x9788000

Playlist: CSE 351 24Sp Lecture Tunes!

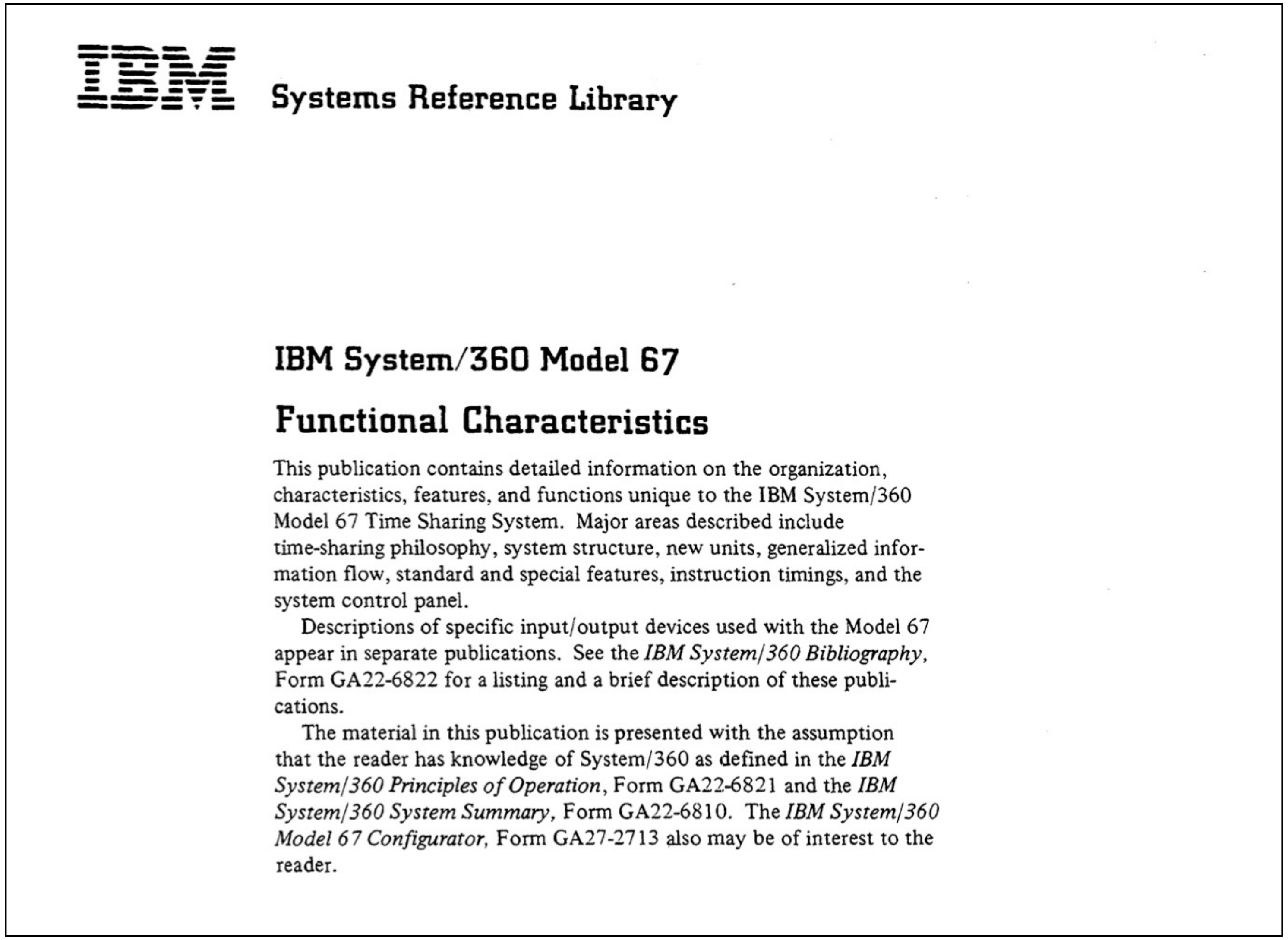
Announcements, Reminders

- Whirlwind week recap & effects
  - Lab 3 & Midterm grades by tonight
  - LC 21 & 22 inked slides coming soon
  - Possible changes still forthcoming!
- HW22 due tonight
  - HW23 due Wednesday (22 May)
  - HW 24/25 due Friday (24 May)
- Lab 4 due tonight by 11:59 PM! Lab 5 due May 31st
- Memorial Day holiday next Monday (27 May)
- Final Exam: June 3rd through June 5th, on Gradescope. More to come!

Reading Review

- Terminology:
  - Address translation: page hit, page fault
  - Translation Lookaside Buffer (TLB): TLB Hit, TLB Miss

But first, some history… (1966)



IBM Systems Reference Library

IBM System/360 Model 67

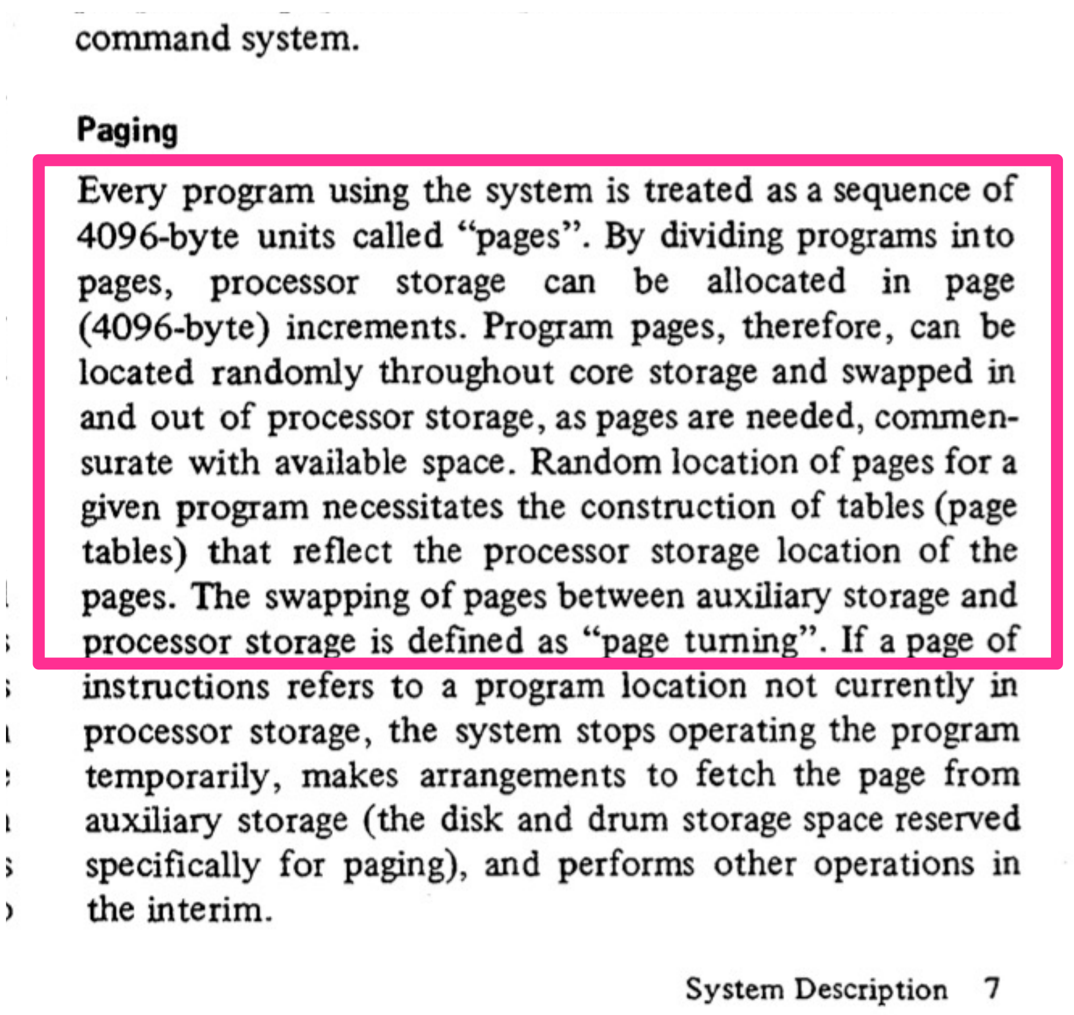
Functional Characteristics

This publication contains detailed information on the organization, characteristics, features, and functions unique to the IBM System/360 Model 67 Time Sharing System. Major areas described include time-sharing philosophy, system structure, new units, generalized information flow, standard and special features, instruction timings, and the system control panel.

Descriptions of specific input/output devices used with the Model 67 appear in separate publications. See the *IBM System/360 Bibliography*, Form GA22-6822 for a listing and a brief description of these publications.

The material in this publication is presented with the assumption that the reader has knowledge of System/360 as defined in the *IBM System/360 Principles of Operation*, Form GA22-6821 and the *IBM System/360 System Summary*, Form GA22-6810. The *IBM System/360 Model 67 Configurator*, Form GA27-2713 also may be of interest to the reader.

But first, some history… (1966)

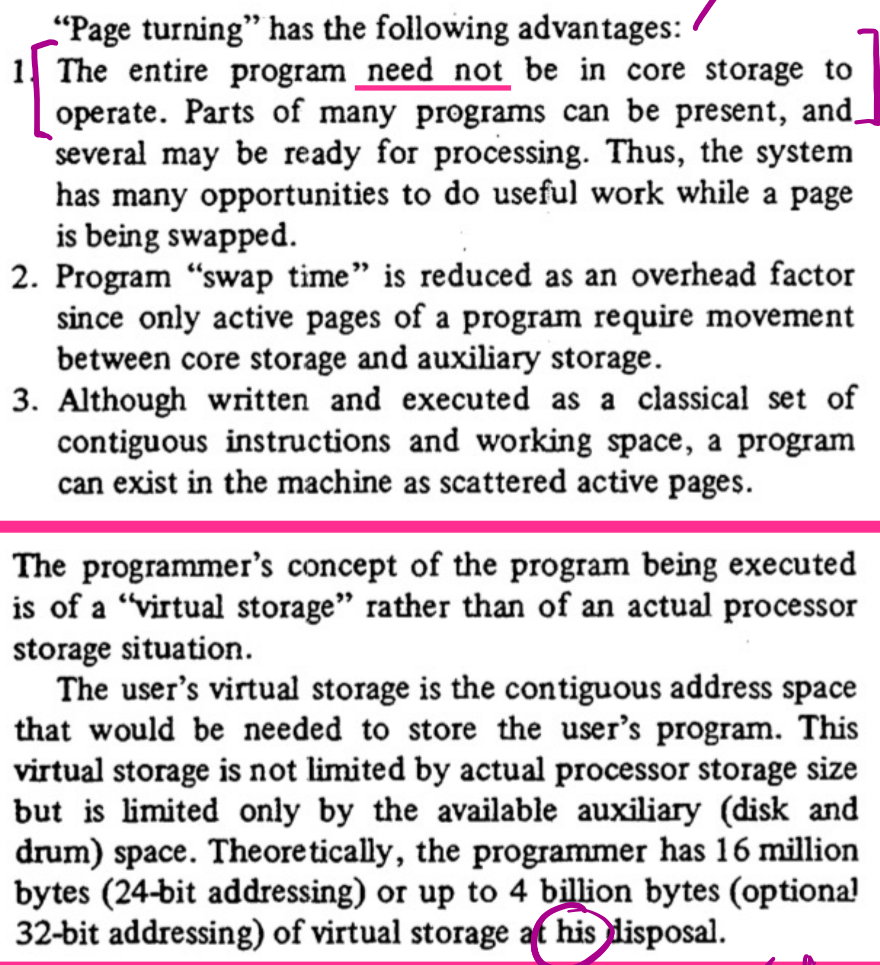


command system.

Paging

Every program using the system is treated as a sequence of 4096-byte units called "pages". By dividing programs into pages, processor storage can be allocated in page (4096-byte) increments. Program pages, therefore, can be located randomly throughout core storage and swapped in and out of processor storage, as pages are needed, commensurate with available space. Random location of pages for a given program necessitates the construction of tables (page tables) that reflect the processor storage location of the pages. The swapping of pages between auxiliary storage and processor storage is defined as "page turning". If a page of instructions refers to a program location not currently in processor storage, the system stops operating the program temporarily, makes arrangements to fetch the page from auxiliary storage (the disk and drum storage space reserved specifically for paging), and performs other operations in the interim.

System Description 7



"Page turning" has the following advantages:

1. The entire program need not be in core storage to operate. Parts of many programs can be present, and several may be ready for processing. Thus, the system has many opportunities to do useful work while a page is being swapped.
2. Program "swap time" is reduced as an overhead factor since only active pages of a program require movement between core storage and auxiliary storage.
3. Although written and executed as a classical set of contiguous instructions and working space, a program can exist in the machine as scattered active pages.

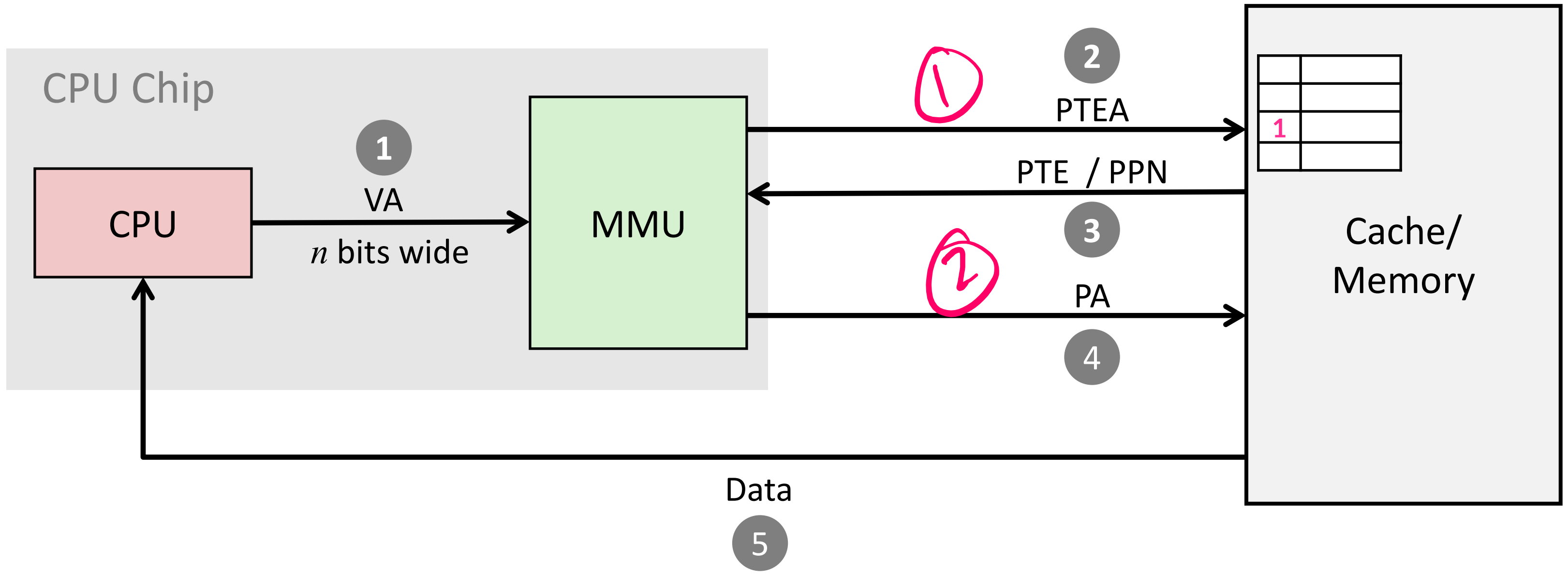
The programmer's concept of the program being executed is of a "virtual storage" rather than of an actual processor storage situation.

The user's virtual storage is the contiguous address space that would be needed to store the user's program. This virtual storage is not limited by actual processor storage size but is limited only by the available auxiliary (disk and drum) space. Theoretically, the programmer has 16 million bytes (24-bit addressing) or up to 4 billion bytes (optional 32-bit addressing) of virtual storage at his disposal.

Allows us to multiprogram!

Rude. Am I a joke to you?

Taking A Step Back: 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; valid bit is one
(Uses PTBR to find beginning of page table for current process)

memory access!

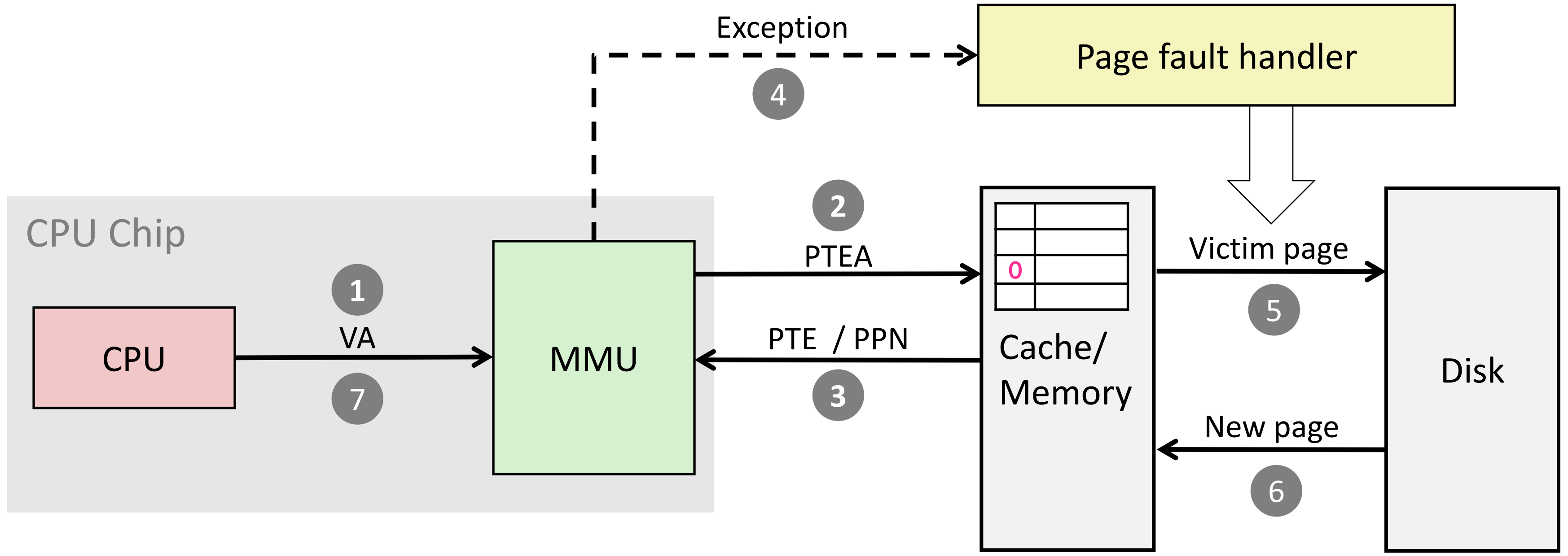
4) MMU sends physical address to cache/memory requesting data

memory access!

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 requested by CPU

Taking a Step Back: 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 <u>zero</u>, so MMU triggers page fault exception & thus page fault handler

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

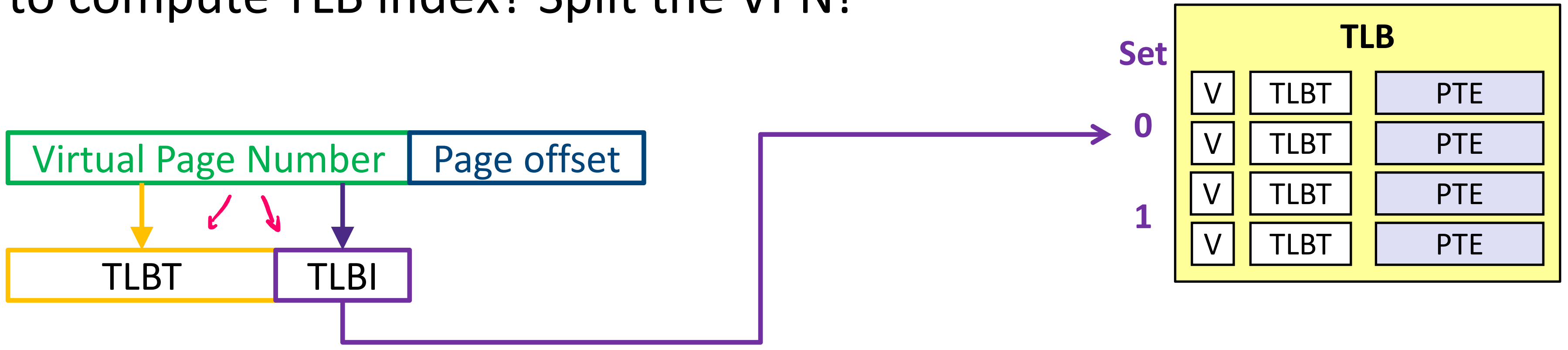
What can we do to make this faster?

Solution: add another cache! 🎉

Cache. Everything.

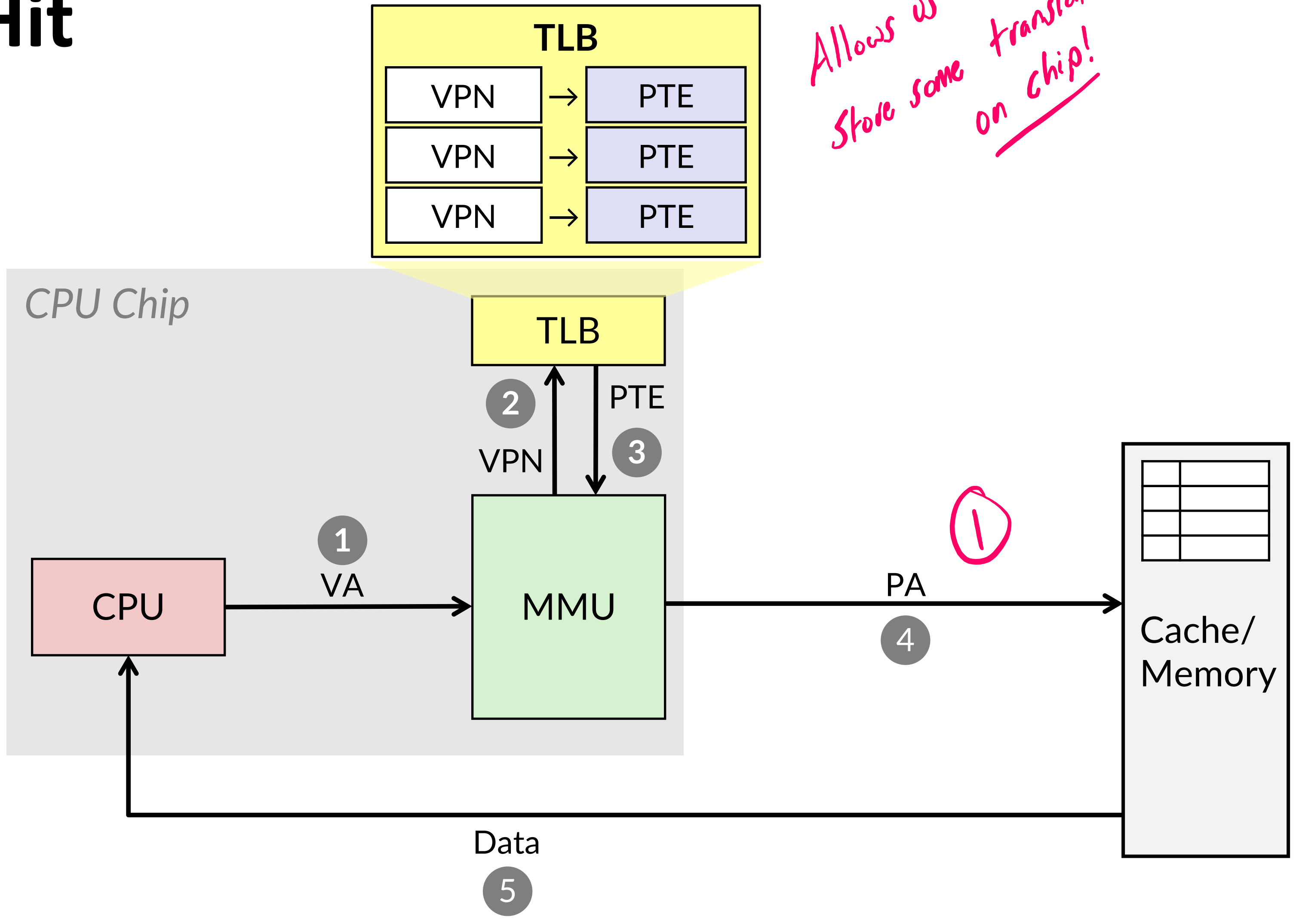
Speeding up Translation with a TLB

- **Translation Lookaside Buffer** (**TLB**):
  - [Stores page table entries for a small number of pages]
    - Modern Intel processors have 128 or 256 entries in 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]
  - Much faster than a page table lookup in cache/memory
  - How to compute TLB index? Split the VPN!



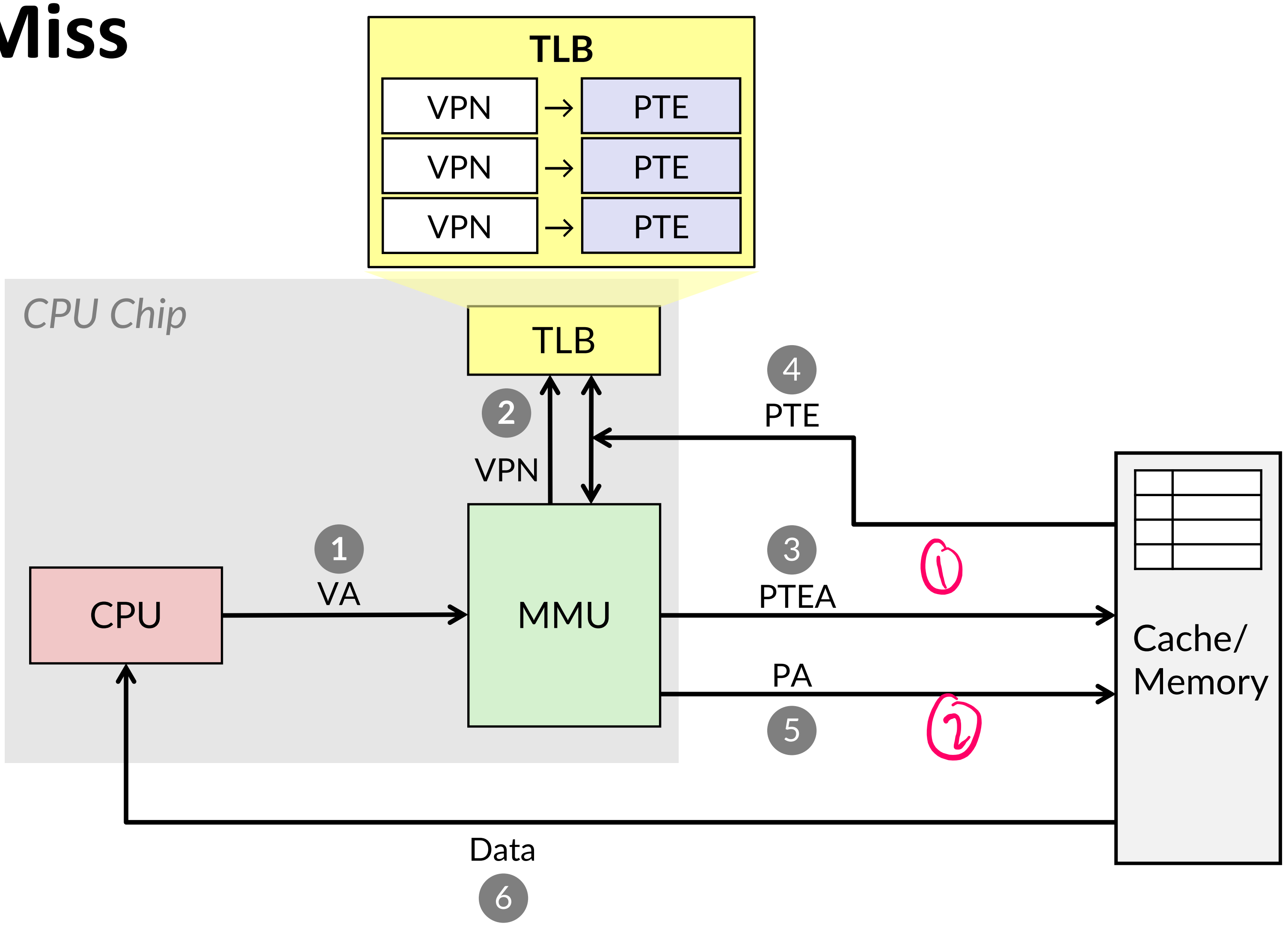
Visual: TLB Hit

Allows us to store some translations on chip!



- A TLB hit eliminates a memory access!

Visual: TLB Miss



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

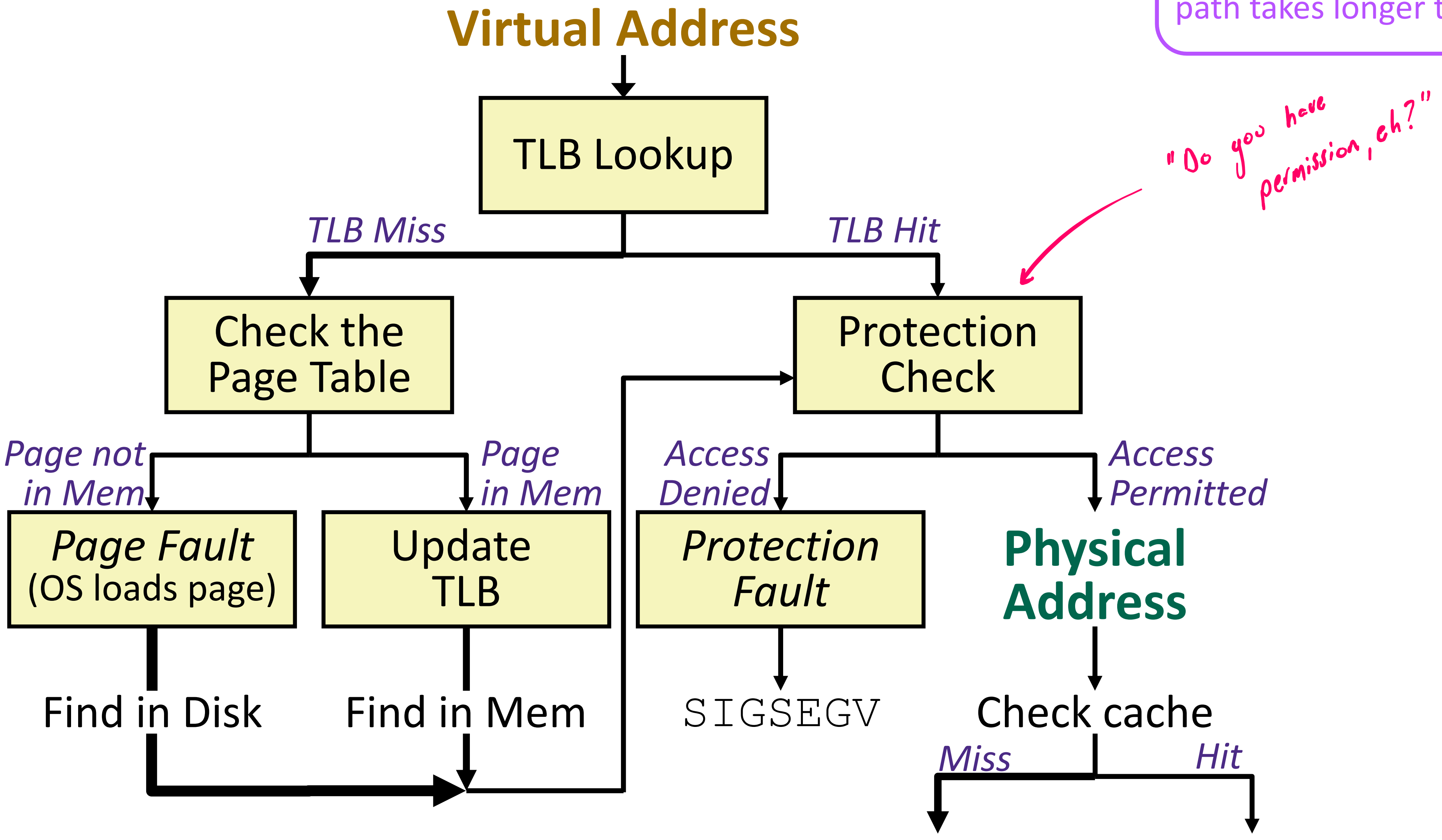
Textual Equivalent: Fetching Data on a Memory Read

1) Address Translation (check TLB)
   - Input: VPN from Virtual Address (VA), Output: PPN for Physical Address (PA)
   - TLB Hit: Fetch translation, return PPN, **DONE!**
   - TLB Miss: Check page table (in memory)
     - Page Table Hit: Load page table entry into TLB, return PPN, **DONE!**
     - Page Fault: Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB, then return PPN, **DONE!**

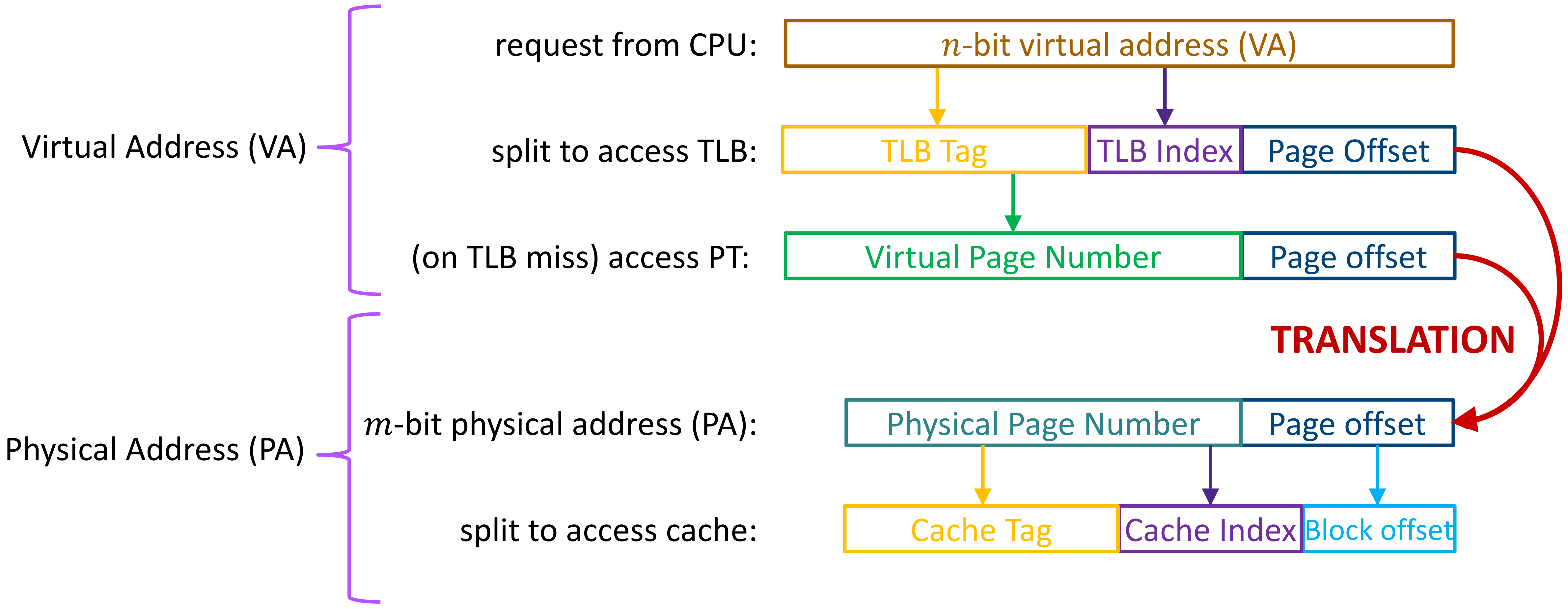
2) Now with PA… Fetch Data (check cache)
   - Input: Physical Address (PA), Output: data
   - Cache Hit: Return data value to processor, **DONE!**
   - Cache Miss: Fetch data value from memory, store it in cache, return it to processor, **DONE!**

Flow Chart Equivalent: Address Translation

NOTE: If the line looks thicker, that means that path takes longer time…



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! 🚨
    - Yep, have to start with a cold TLB 🥶 Boo!
  - Cache:
    - Can leave alone because storing based on PAs—good for shared data

Summary of Address Translation Symbols

- Basic Parameters

$\mathrm{N} = 2^n$	Number of addresses in virtual address space
$\mathrm{M} = 2^m$	Number of addresses in physical address space
$\mathrm{P} = 2^p$	Page size (bytes)

- Components of the virtual address (VA)

VPO	Virtual page offset
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag

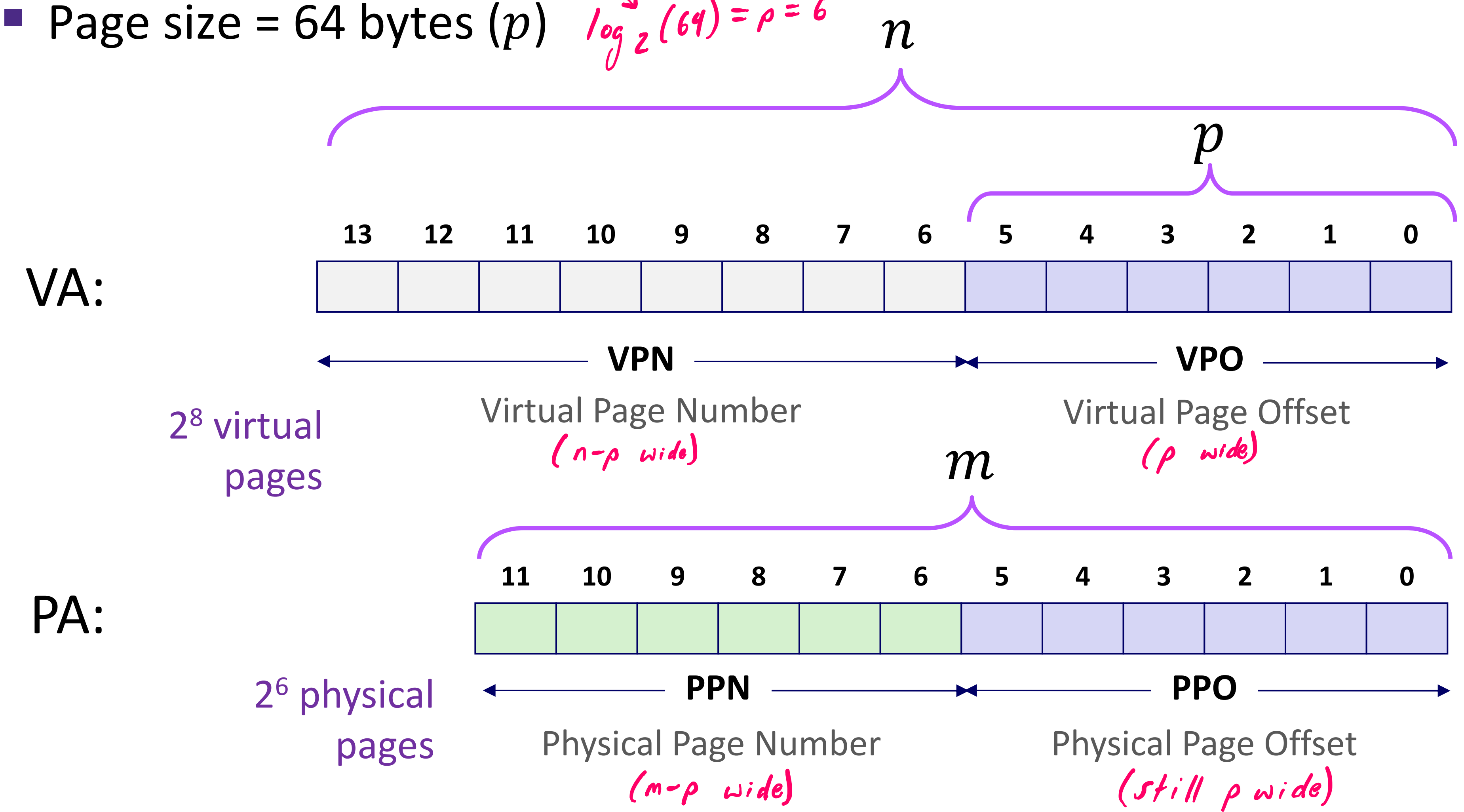
- Components of the physical address (PA)

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

And now let’s put it all together…

Simple Memory System Example

- Addressing: 14-bit virtual addresses (n) & 12-bit physical address (m)
  - Page size = 64 bytes (p) $\log_2(64) = p = 6$



Simple Memory System: Page Table

- Only showing first 16 entries (out of $\underline{2^8}$)
  - Size of page table is 2^{n-p} entries, remember!
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

$2^{n-p} \Rightarrow 2^{14-6} = 2^8$

one for each <u>virtual</u> page

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

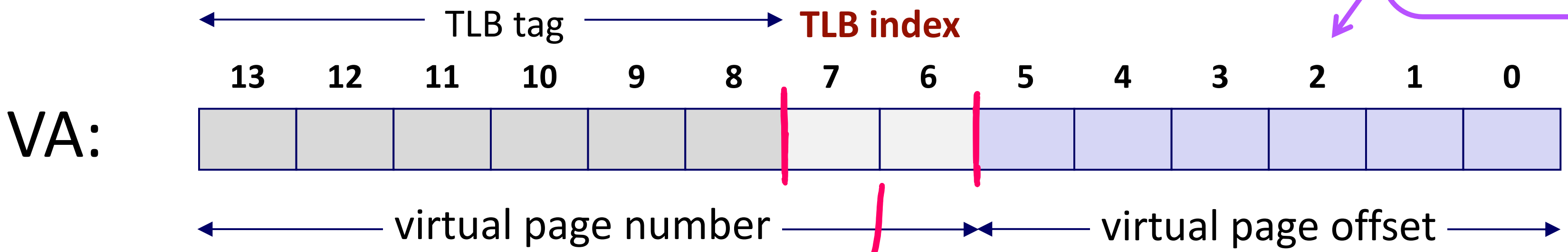
- 16 entries total
- 4-way set associative

$\Rightarrow$ $16/4 = 4$ sets, each of 4 ways

$\therefore$ 2 bits for set index

Why does the TLB ignore the page offset?
It's not its job to care!

Way fewer than 2^8!

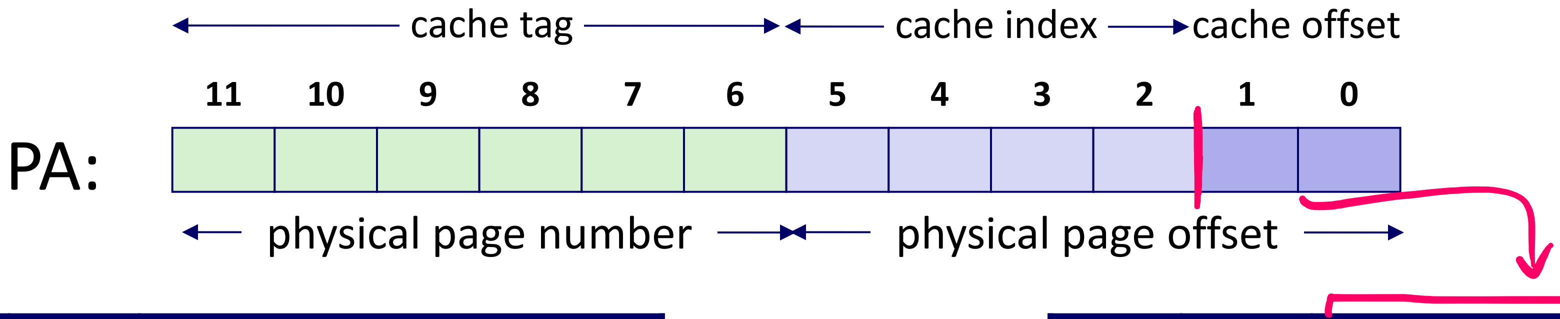


Set	Way 0			Way 1			Way 2			Way 3		
	Tag	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

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 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:

① Miss!

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

① Hit!

① Miss!

① Hit!

Thus can go straight to cache!

Thus check PT using VPN

Page table (partial):

② Hit!

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

② Miss!

Thus check PT using VPN

Cache:

②

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
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

③ Miss! (wrong tag...)

Index	Tag	V	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	–	–	–	–

③ Hit!

Memory Request Example #1

[redacted]

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 0xF TLBT 0x3 TLBI 0x3 TLB Hit? Y Page Fault? N PPN 0x0D

- Physical Address:

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0
PPN						PPO					

CT 0x0D CI 0x5 CO 0x0 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`

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 0xE TLBT 0x3 TLBI 0x2 TLB Hit? N Page Fault? Y PPN ???

- Physical Address:

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Can't go any further!

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: bits 13–8; TLBI: bits 7–6

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: bits 13–6; VPO: bits 5–0

VPN 0x0 TLBT 0x0 TLBI 0x0 TLB Hit? N Page Fault? N PPN 0x28

- Physical Address:

CT: bits 11–6; CI: bits 5–2; CO: bits 1–0

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

PPN: bits 11–6; PPO: bits 5–0

CT 0x28 CI 0x8 CO 0x0 Cache Hit? N Data (byte) ???

Can't go further!

Memory Request Example #4

[redacted]

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

- Virtual Address: `0x036B`

TLBT: bits 13–8; TLBI: bits 7–6; VPN: bits 13–6; VPO: bits 5–0

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 0xD TLBT 0x3 TLBI 0x1 TLB Hit? Y Page Fault? N PPN 0x2D

- Physical Address:

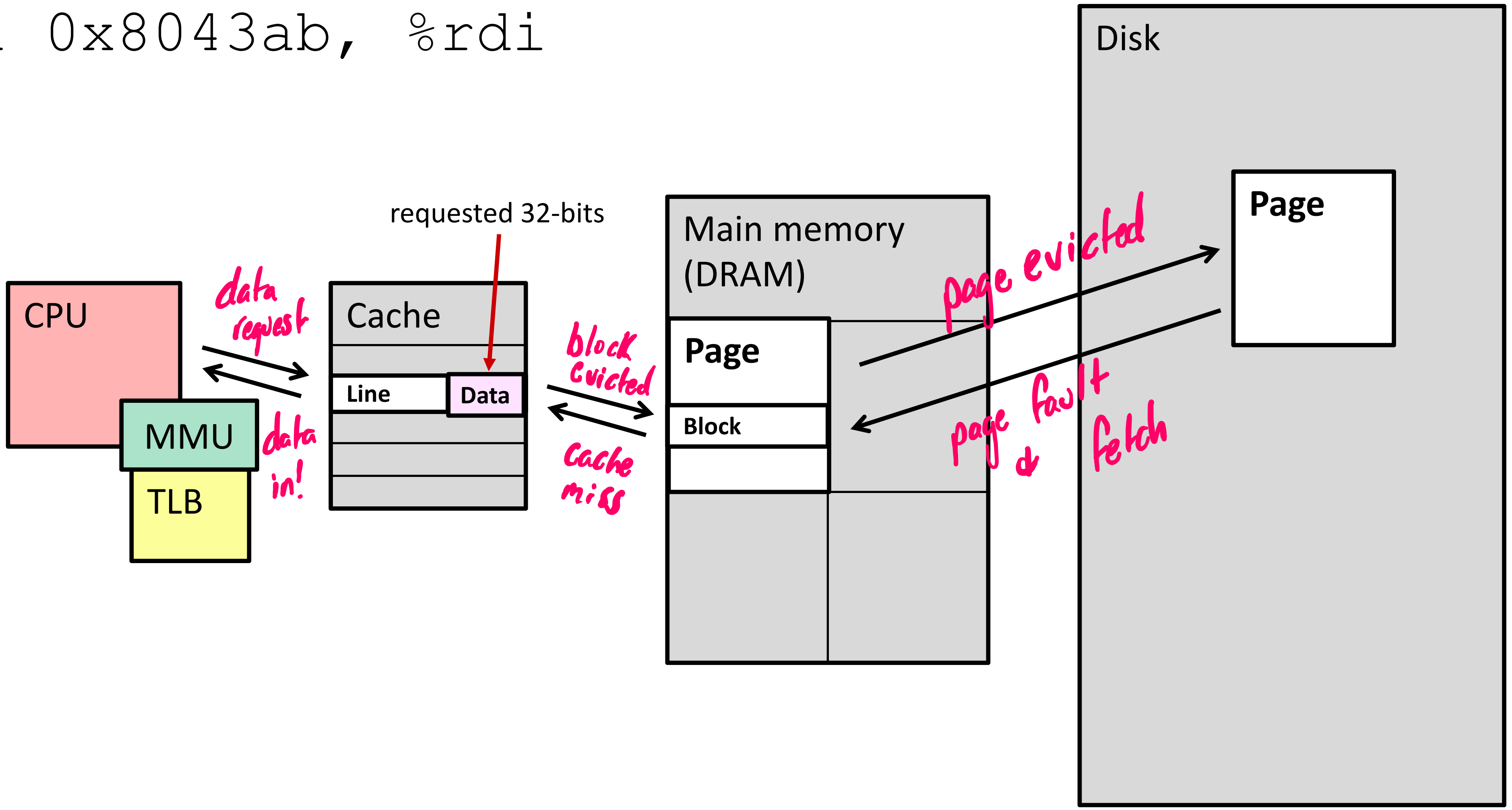
CT: bits 11–6; CI: bits 5–2; CO: bits 1–0; PPN: bits 11–6; PPO: bits 5–0

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

CT 0x2D CI 0xA CO 0x3 Cache Hit? Y Data (byte) 0x3B

Memory Overview (Data Flow)

- `movl 0x8043ab, %rdi`



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

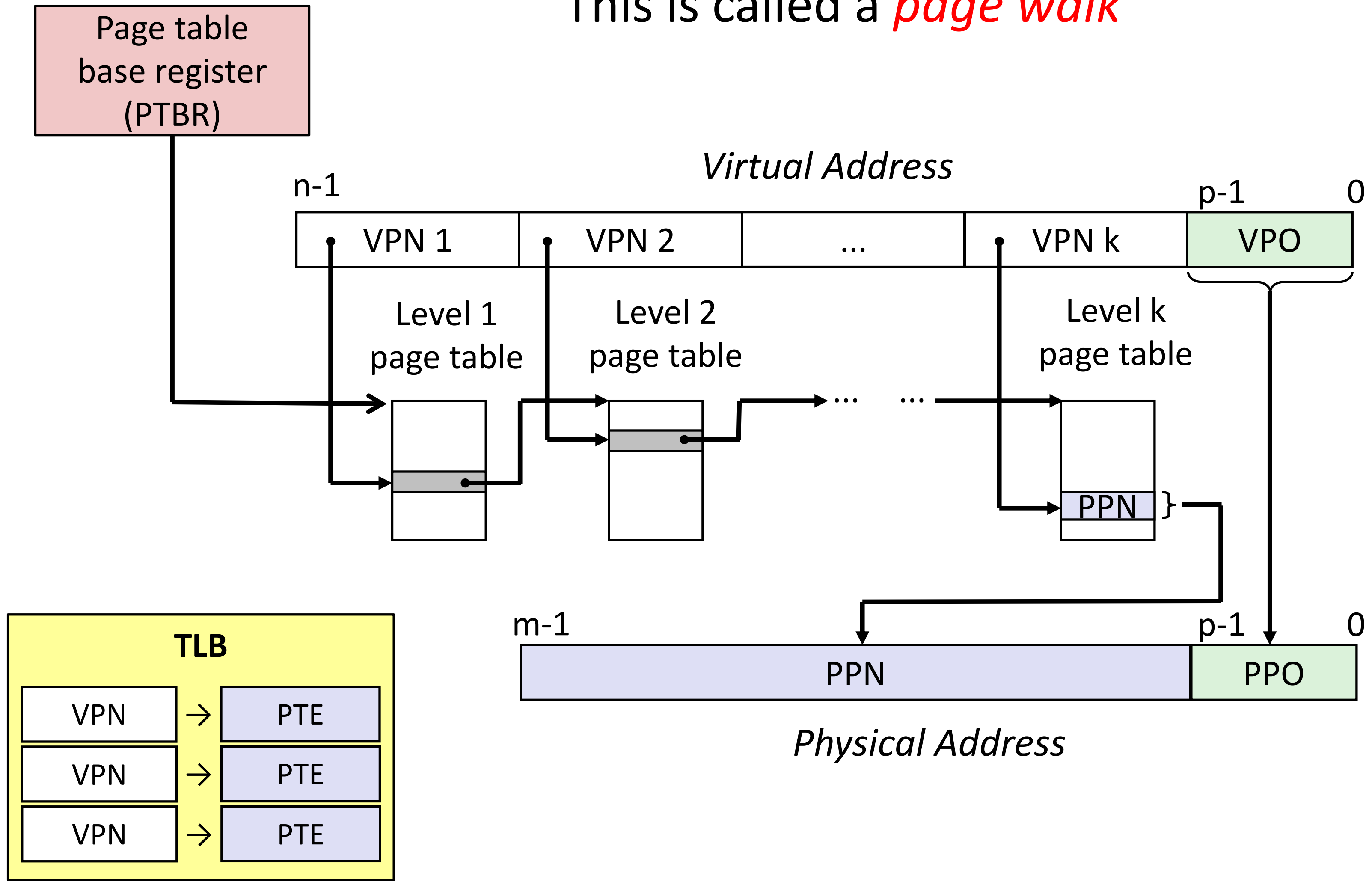
This is extra (non-testable) material

- 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*



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