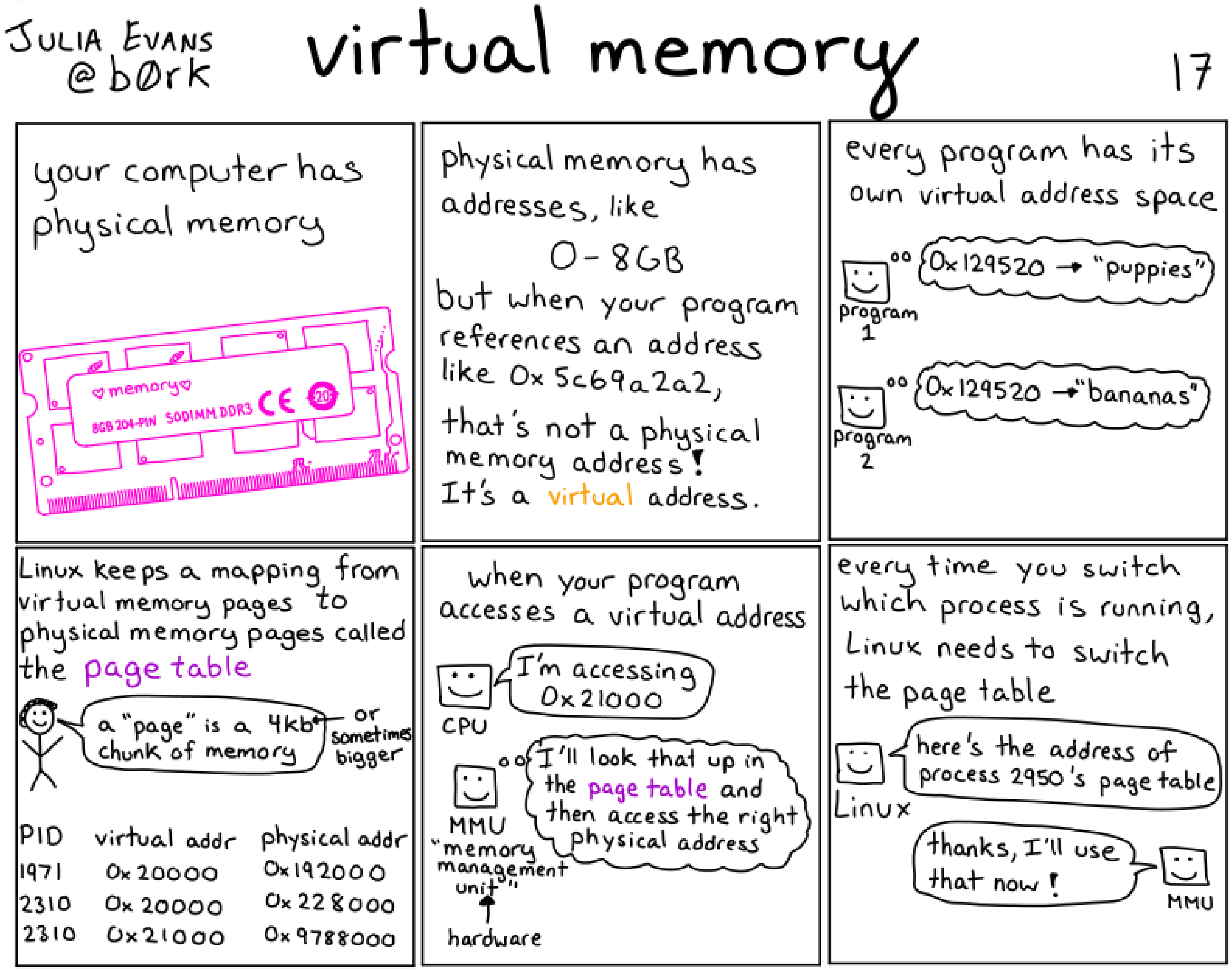
# Virtual Memory III

CSE 351 Autumn 2024

**Instructor:**
Ruth Anderson

**Teaching Assistants:**
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub




https://wizardzines.com/comics/virtual-memory/

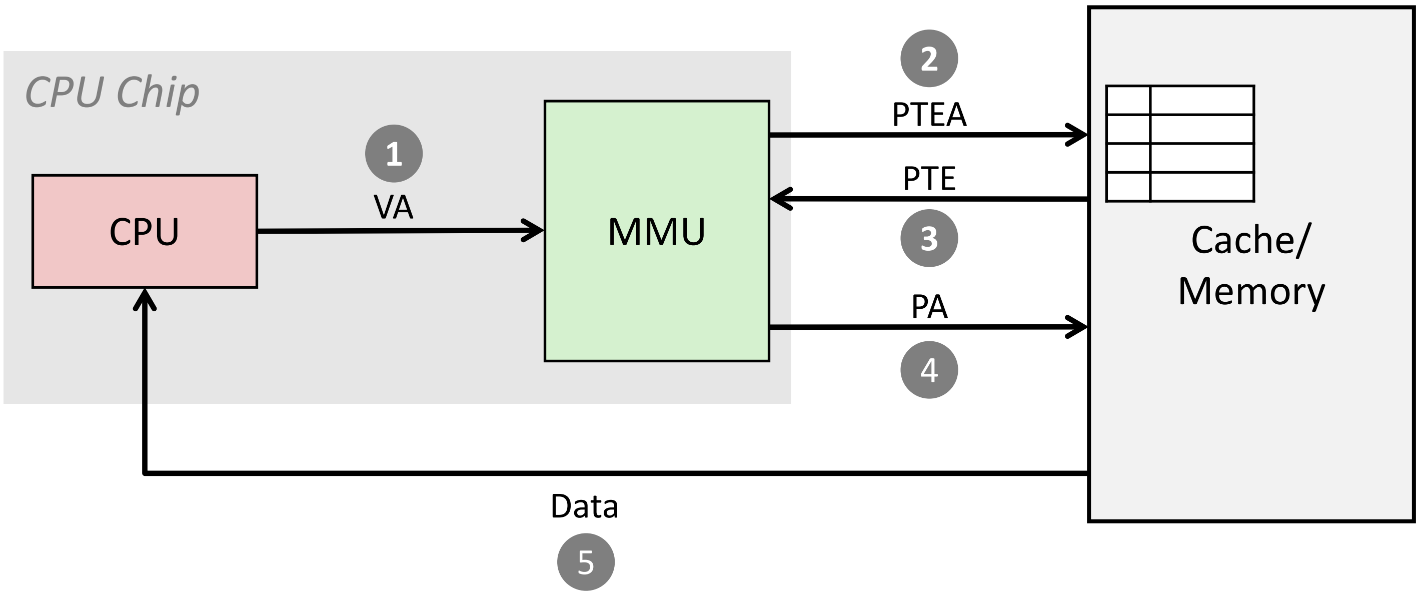
# Relevant Course Information

- HW23 due Wednesday (11/27) @ 11:59 pm
- No HW24
- HW25 due Monday (12/02) @ 11:59 pm
- HW26 due Wednesday (12/04) @ 11:59 pm
- Lab 5 (on Mem Alloc) due Thurs (12/05) @ 11:59pm
  - The most significant amount of C programming you will do in this class – combines lots of topics from this class: pointers, bit manipulation, structs, examining memory
  - Understanding the concepts *first* and efficient *debugging* will save you lots of time
  - Light style grading

# Reading Review

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

# Address Translation: Page Hit




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

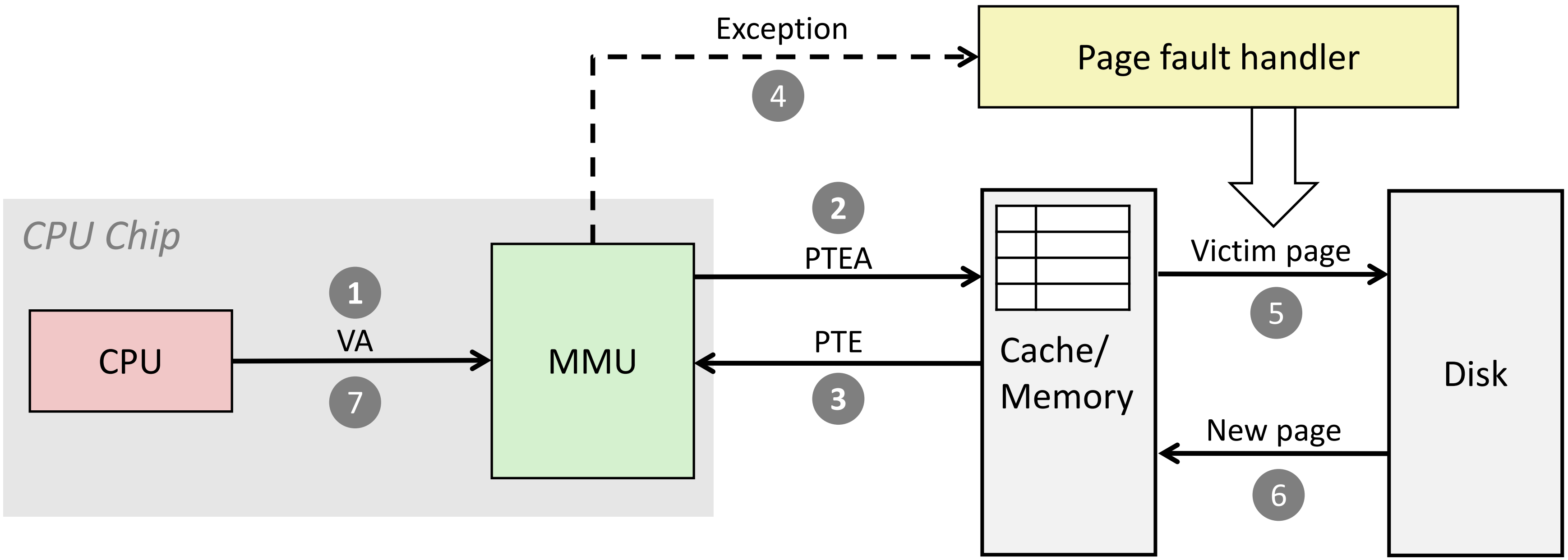
**2-3)** MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

**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

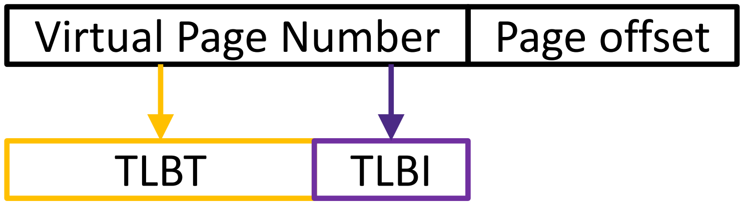
- The MMU accesses memory *twice*:
  - once to get the PTE for translation, and
  - then again for the actual memory request

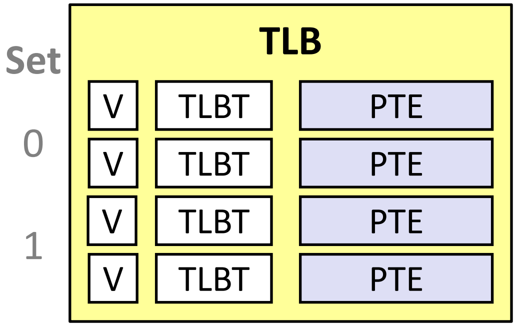
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

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

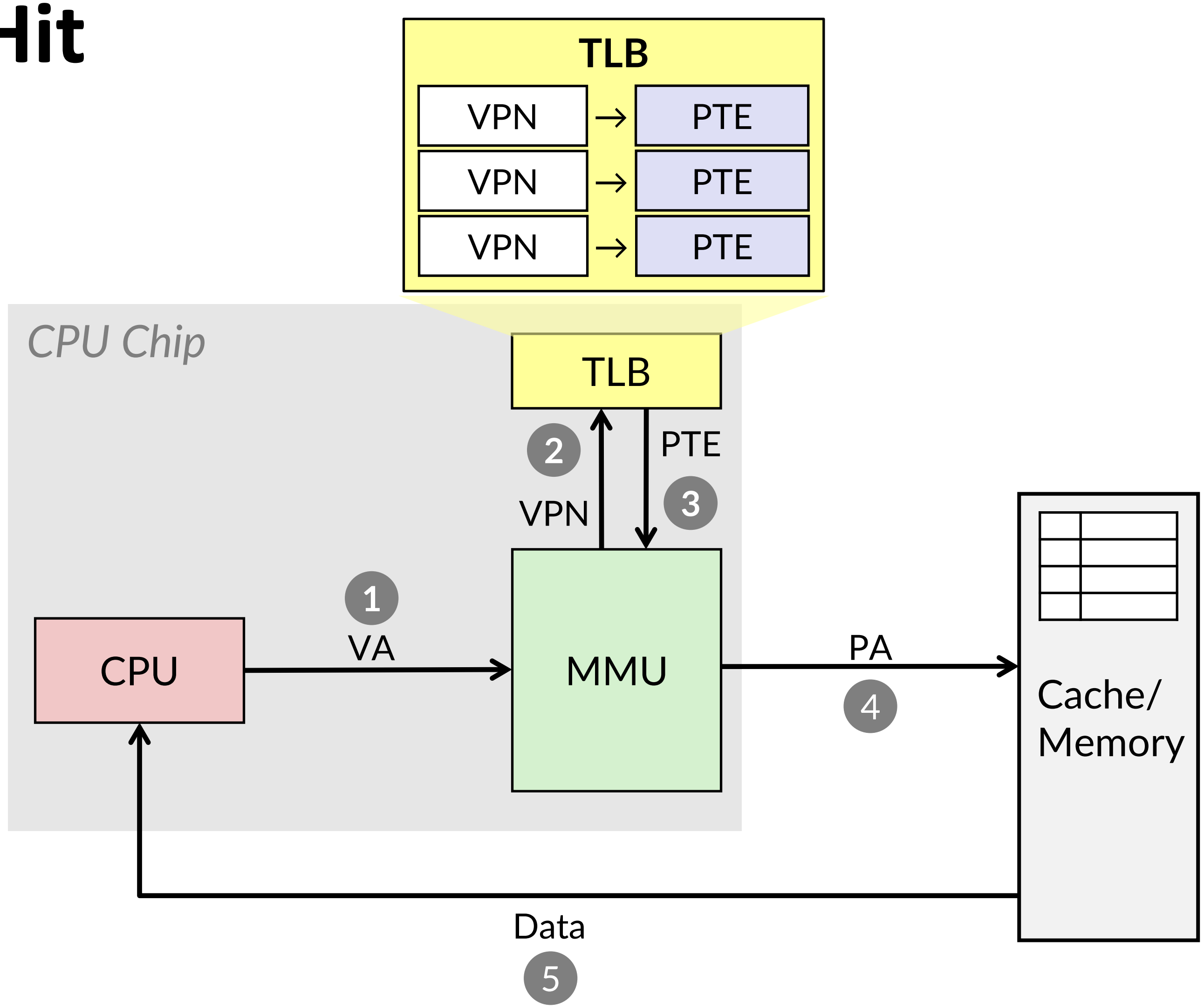
# Speeding up Translation with a TLB

- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
  - Stores *page table entries* for a small number of pages
    - Maps virtual page numbers to physical page numbers
    - Modern Intel processors have 128 or 256 entries in TLB
    - Much faster than a page table lookup in cache/memory
  - Where should a PTE go in the TLB?
    - Split VPN into **TLB Tag** and **TLB Index** based on # of sets in TLB



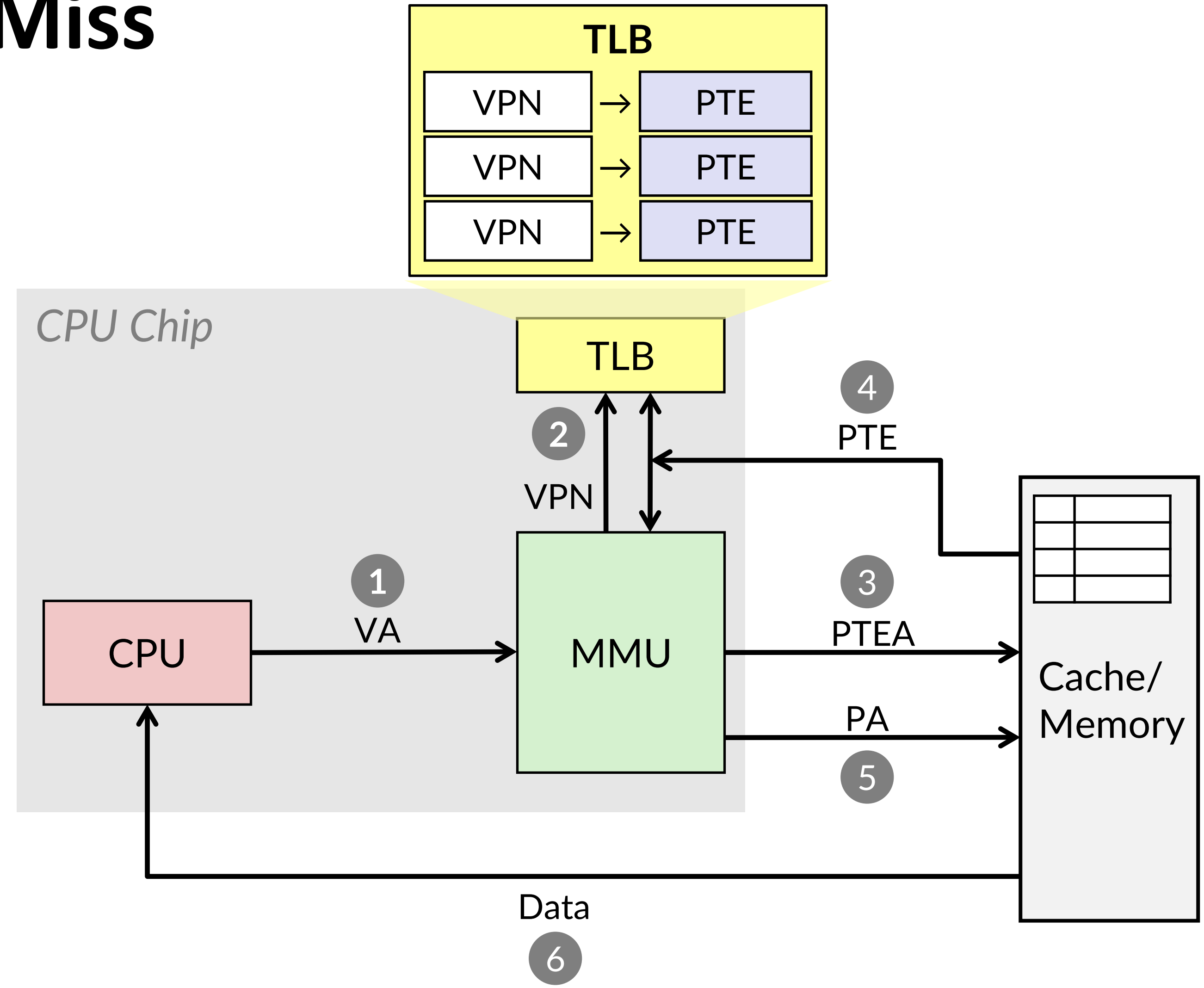



# TLB Hit




- A TLB hit eliminates a memory access!

# TLB Miss

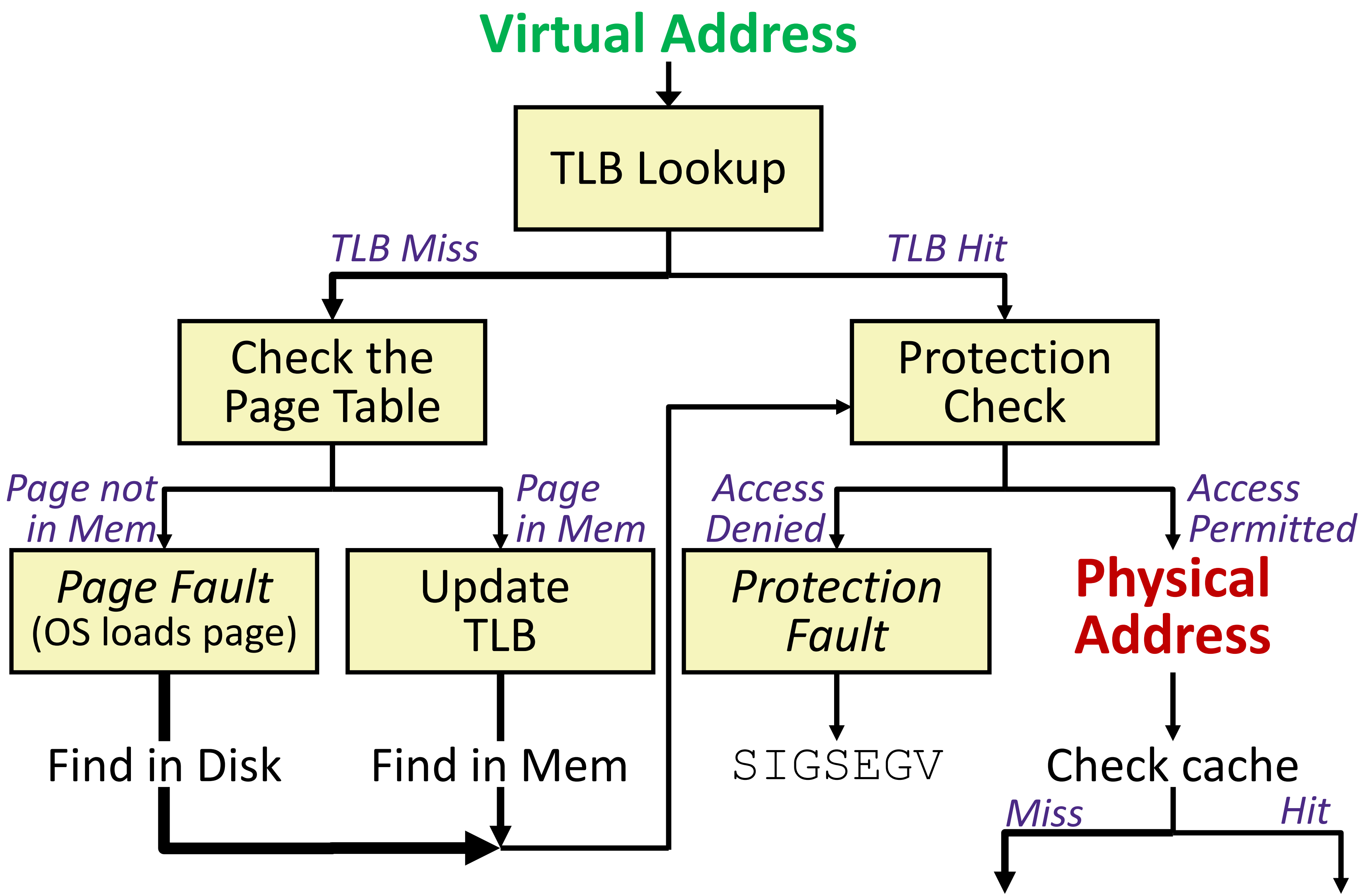



- A TLB miss incurs an additional memory access (the PTE)
  - Fortunately, TLB misses are rare

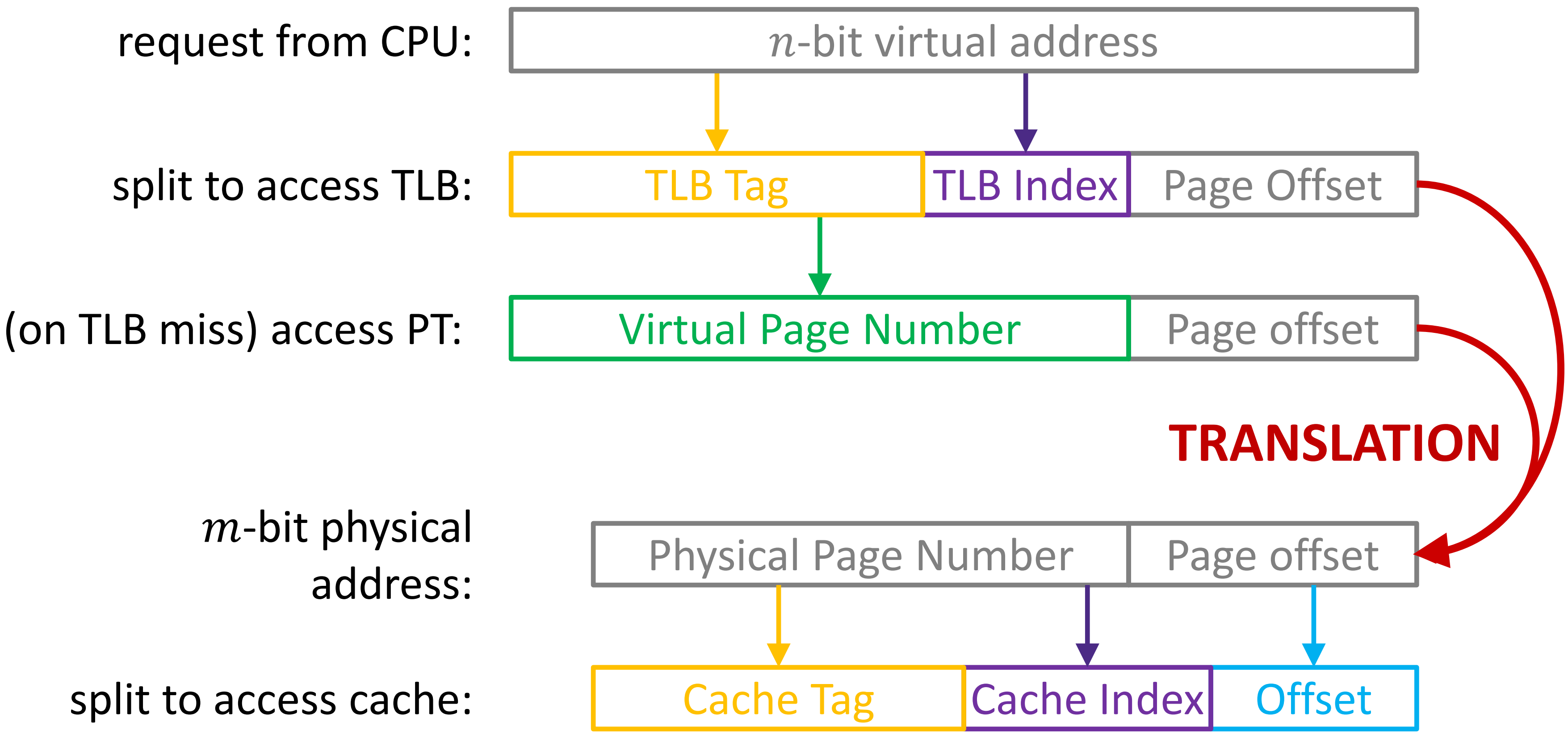
# Summary: Fetching Data on a Memory Read

1) Address Translation (check TLB)
   - Input: VPN, Output: PPN
   - *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 that we have a 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!**

# Address Translation Flow Chart



# Address Manipulation



# Context Switching Revisited

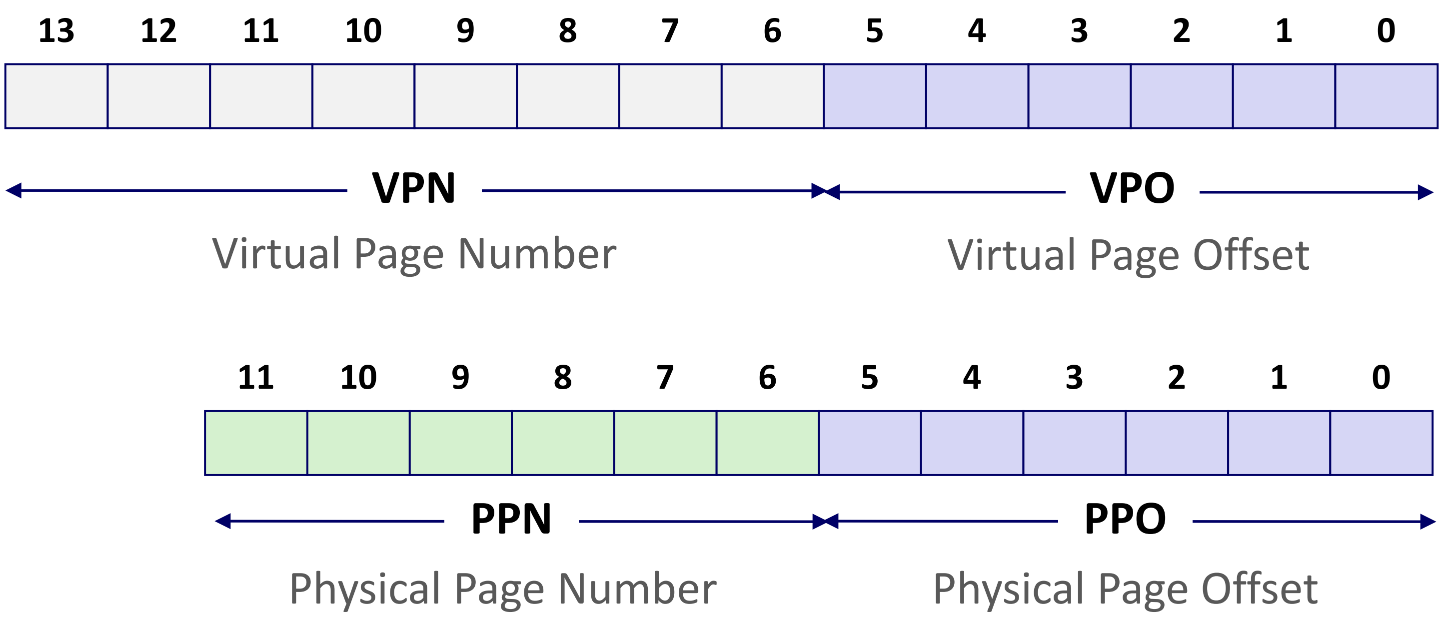
- What needs to happen when the CPU switches processes?
  - Registers:
    - Save state of old process, load state of new process
    - Including the Page Table Base Register (PTBR)
  - Memory:
    - Nothing to do! Pages for processes already exist in memory/disk and protected from each other
  - TLB:
    - *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
  - $\mathrm{N} = 2^n$ 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

# Simple Memory System Example (small)

- Addressing
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes



# Simple Memory System: Page Table

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

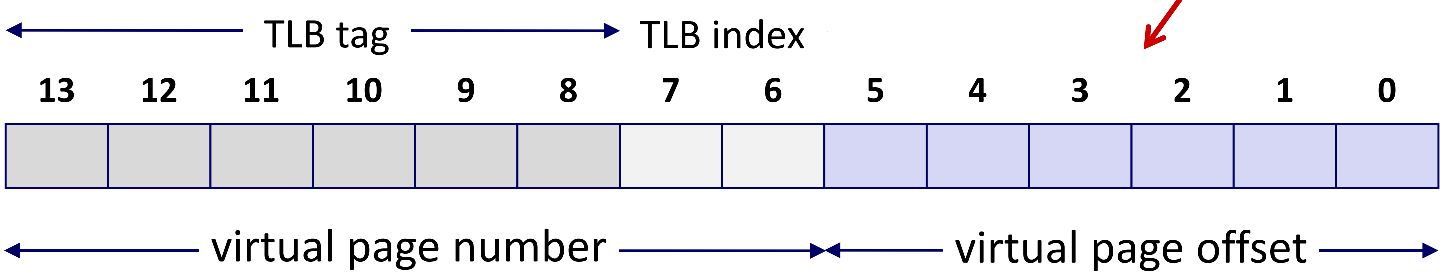
| *VPN* | *PPN* | *Valid* |
|---|---|---|
| **0** | 28 | 1 |
| **1** | – | 0 |
| **2** | 33 | 1 |
| **3** | 02 | 1 |
| **4** | – | 0 |
| **5** | 16 | 1 |
| **6** | – | 0 |
| **7** | – | 0 |

| *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

Why does the TLB ignore the page offset?




| 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

← cache tag →← cache index →cache offset

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

← physical page number →← physical page offset →

| 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:

| 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 |

## 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 |
|---|---|---|---|---|---|---|
| 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 | 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 | – | – | – | – |

# Memory Request Example #1

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

- Virtual Address: `0x03D4`

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 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

VPN (bits 13–6) | VPO (bits 5–0)

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

- 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

PPN (bits 11–6) | PPO (bits 5–0)

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

# 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 ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

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

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 | TLBI

| 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 | VPO

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

- Physical Address:

CT | CI | CO

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

PPN | PPO

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`

← TLBT → ← TLBI →

| 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 → ← VPO →

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

- Physical Address:

← CT → ← CI → ← CO →

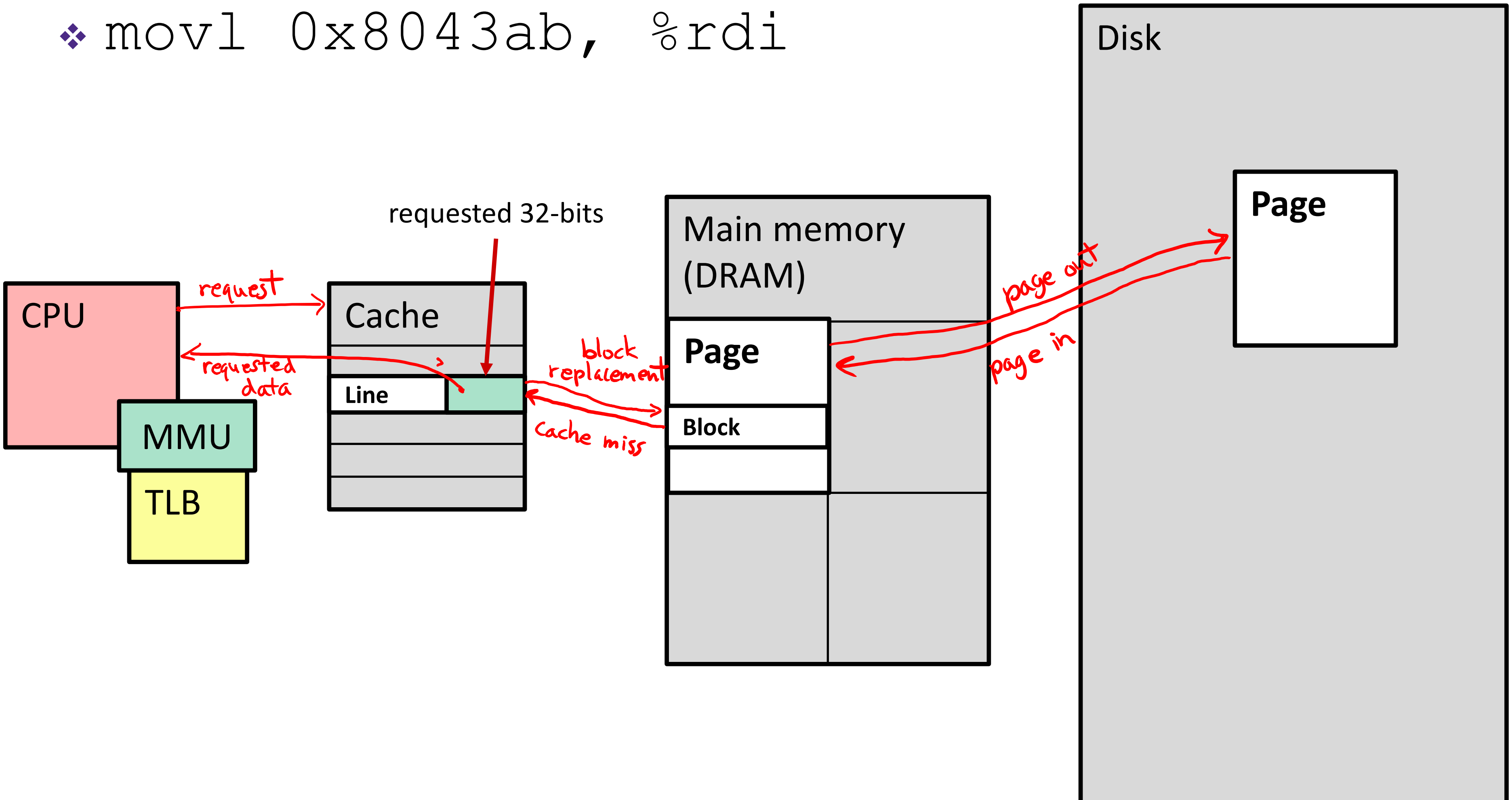
| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |

← PPN → ← PPO →

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

# 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:
  - $n = 64$ 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 ($m-p$) + management bits (V,D,R,W,X) = 20+5 = 25 bits ≈ 3 bytes

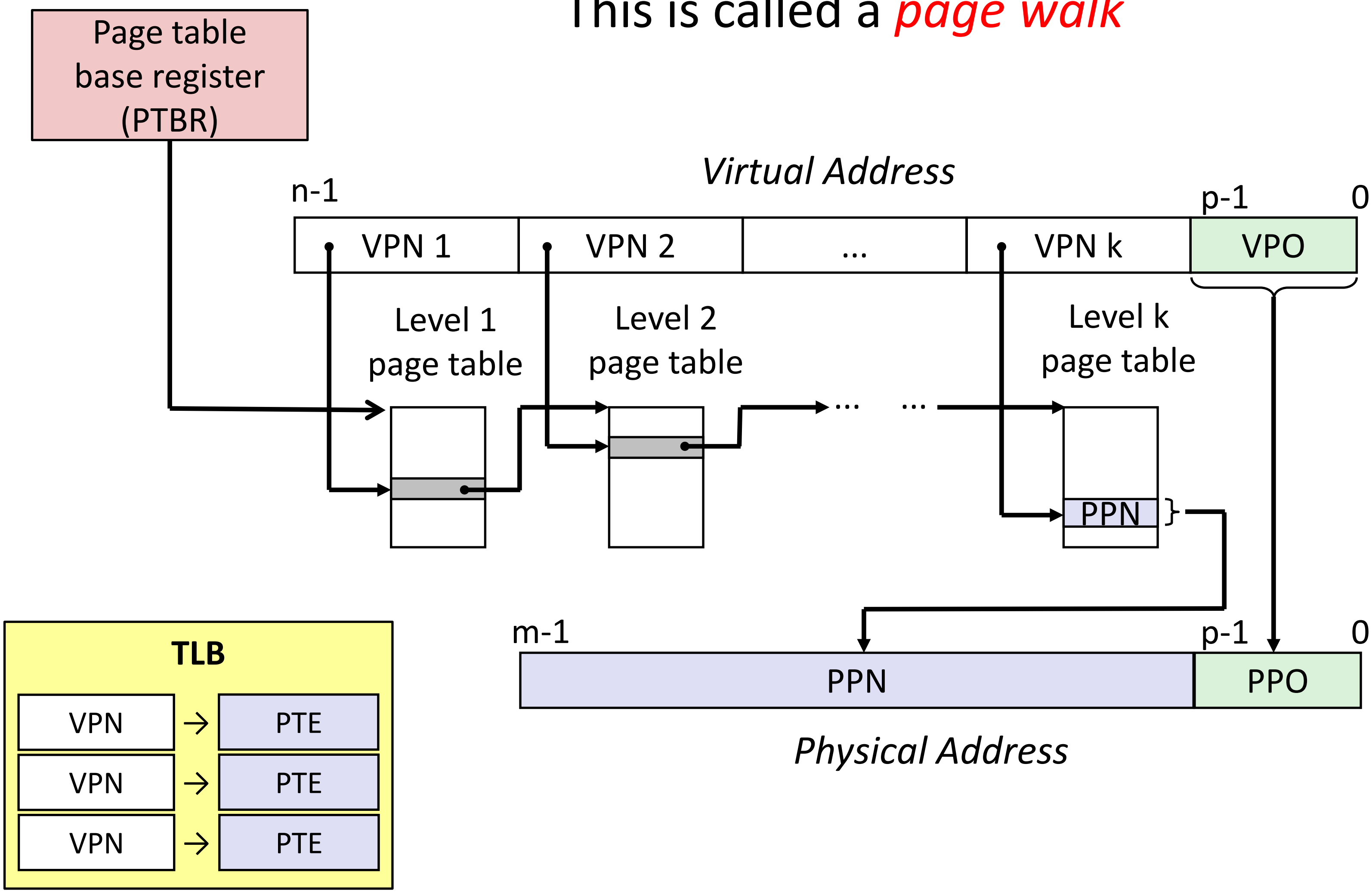
  $\approx 2^{52} + 2^{51}$ bytes per page table!

  - **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

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