

# Virtual Memory II

CSE 351 Winter 2018

## Instructor:

Mark Wyse

## Teaching Assistants:

Kevin Bi

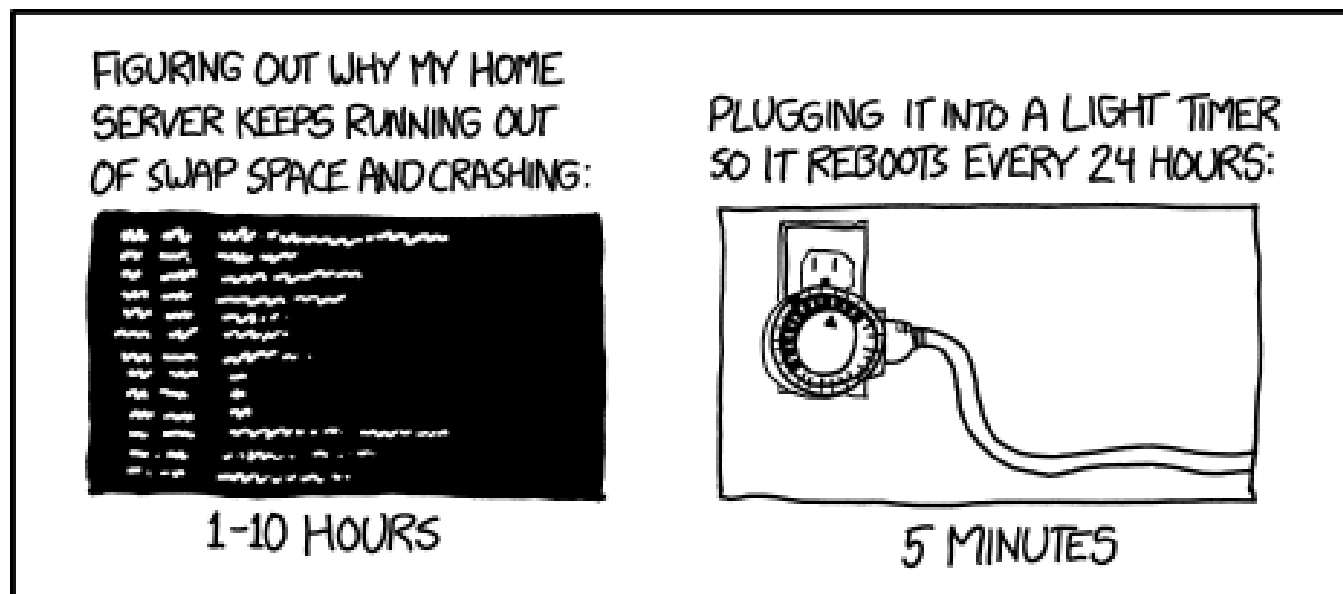
Parker DeWilde

Emily Furst

Sarah House

Waylon Huang

Vinny Palaniappan



WHY EVERYTHING I HAVE IS BROKEN

<https://xkcd.com/1495/>

# Administrative

- ❖ Lab 4 due Wednesday by 11:59 pm!
- ❖ Homework 5 released today, due 3/7
  - Processes and Virtual Memory

# Virtual Memory (VM)

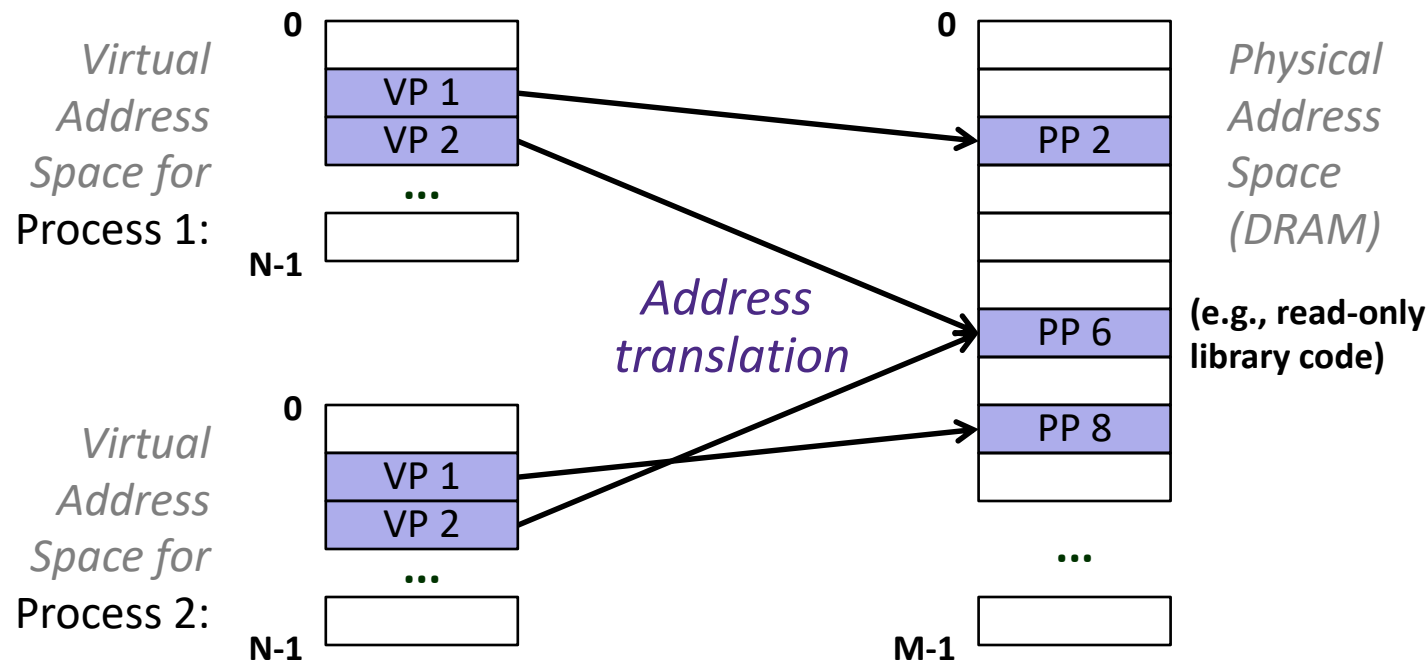
- ❖ Overview and motivation
- ❖ VM as a tool for caching
- ❖ Address translation
- ❖ **VM as a tool for memory management**
- ❖ **VM as a tool for memory protection**

# Review: Terminology

- ❖ Context switch
  - Switch between processes on the same CPU
- ❖ Page in
  - Move pages of virtual memory from disk to physical memory
- ❖ Page out
  - Move pages of virtual memory from physical memory to disk
- ❖ Thrashing
  - Total working set size of processes is larger than physical memory and causes excessive paging in and out instead of doing useful computation

# VM for Managing Multiple Processes

- ❖ Key abstraction: each process has its own virtual address space
  - It can view memory as *a simple linear array*
- ❖ With virtual memory, this simple linear virtual address space **need not be contiguous in physical memory**
  - Process needs to store data in another VP? Just map it to *any* PP!



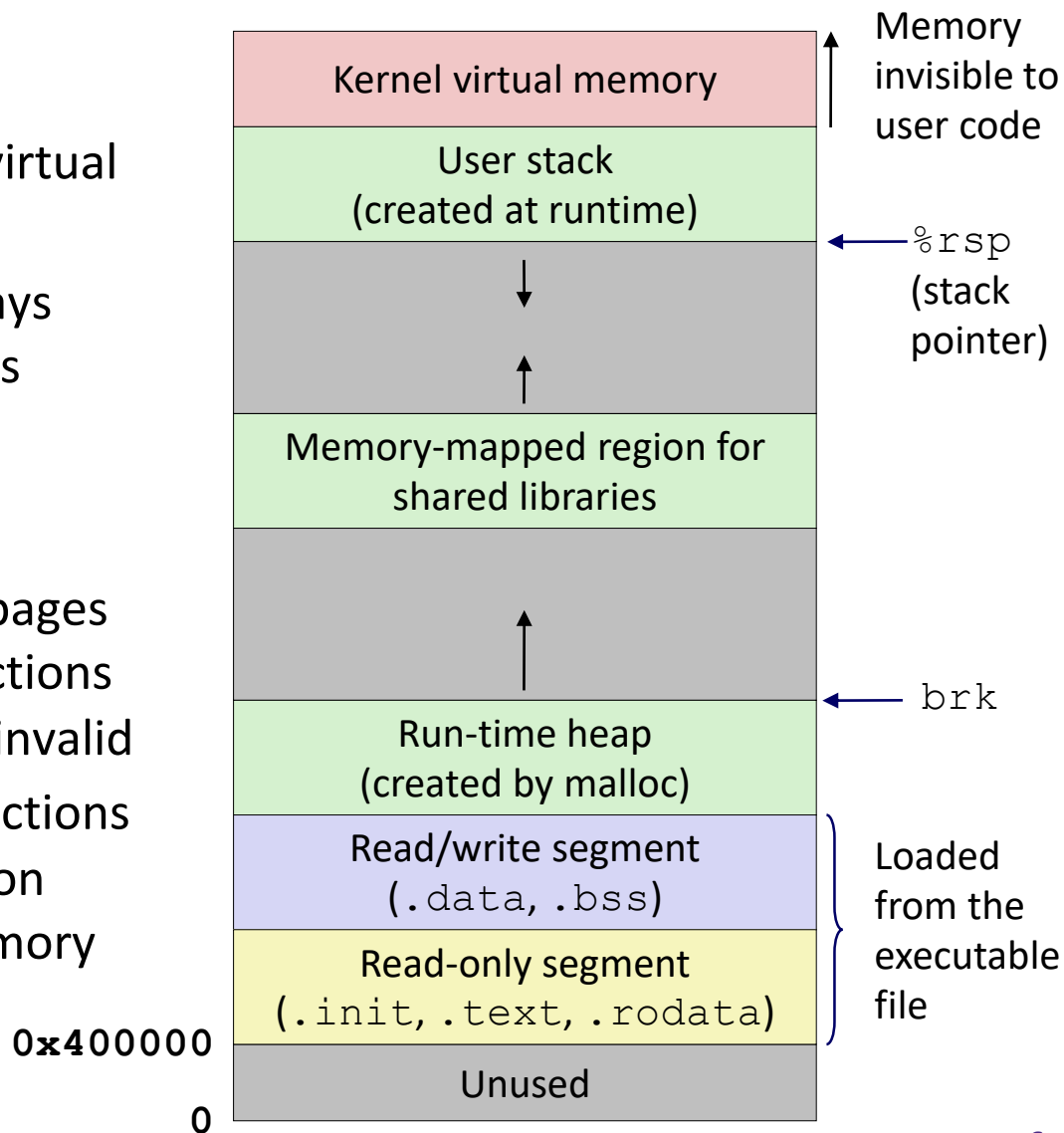
# Simplifying Linking and Loading

## ❖ Linking

- Each program has similar virtual address space
- Code, Data, and Heap always start at the same addresses

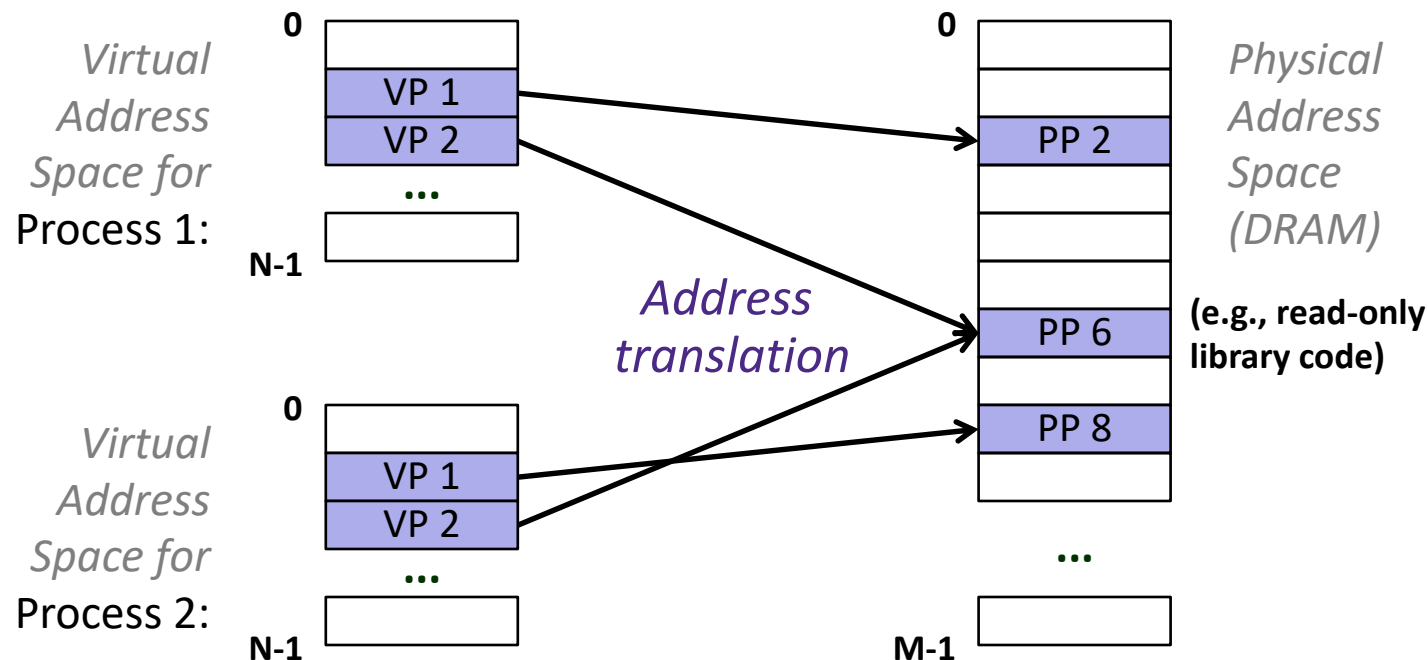
## ❖ Loading

- `execve` allocates virtual pages for `.text` and `.data` sections & creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



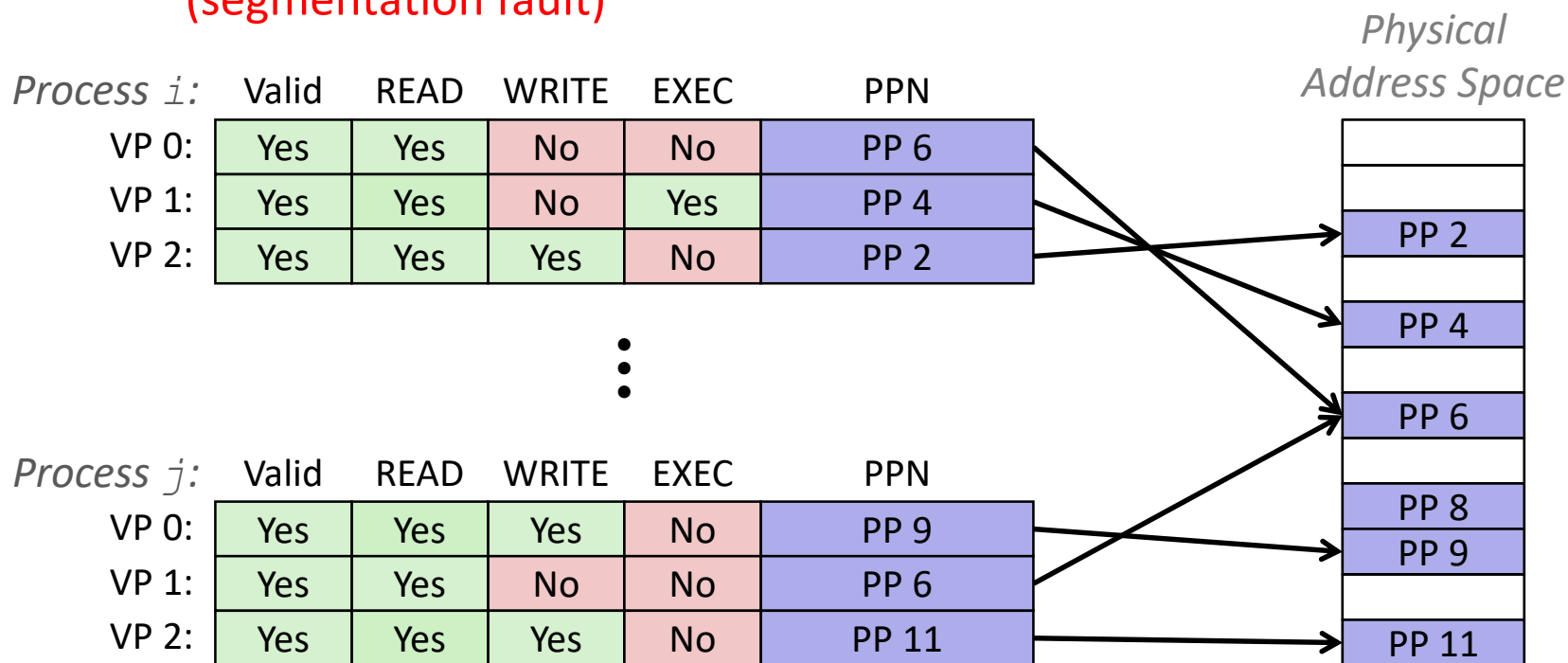
# VM for Protection and Sharing

- ❖ The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes
  - **Sharing:** map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection:** process can't access physical pages to which none of its virtual pages are mapped (here: Process 2 can't access PP 2)



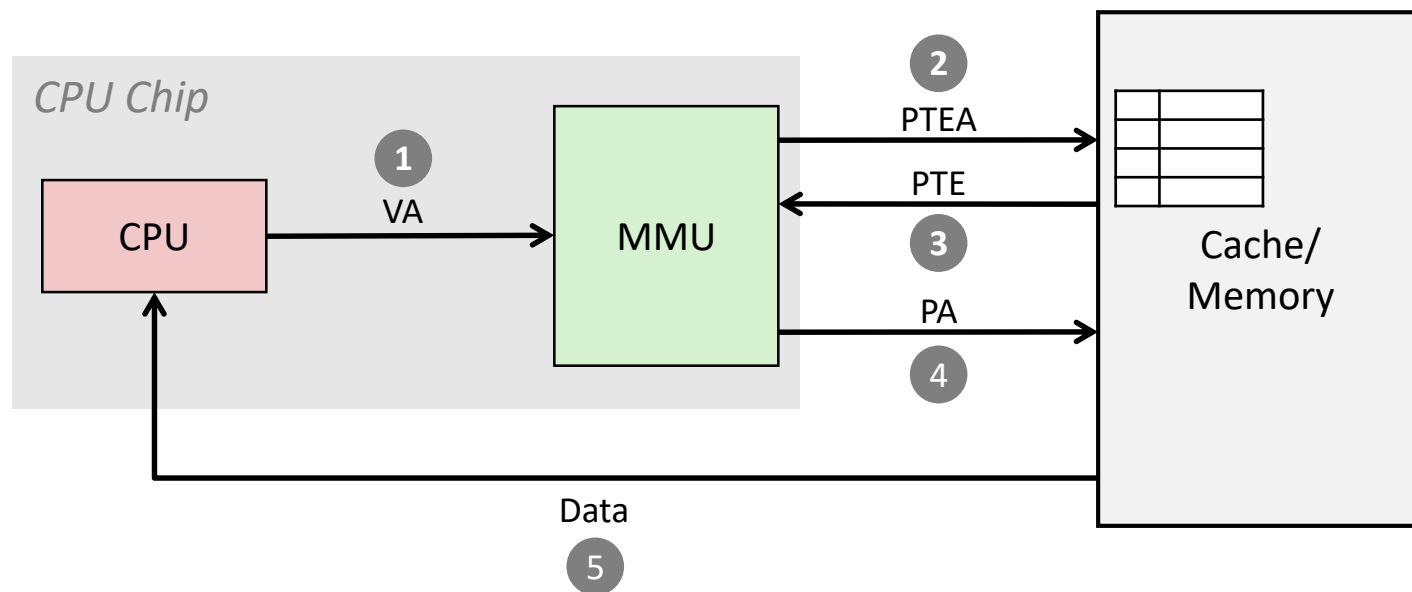
# Memory Protection Within Process

- ❖ VM implements read/write/execute permissions
  - Extend page table entries with permission bits
  - MMU checks these permission bits on every memory access
    - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)





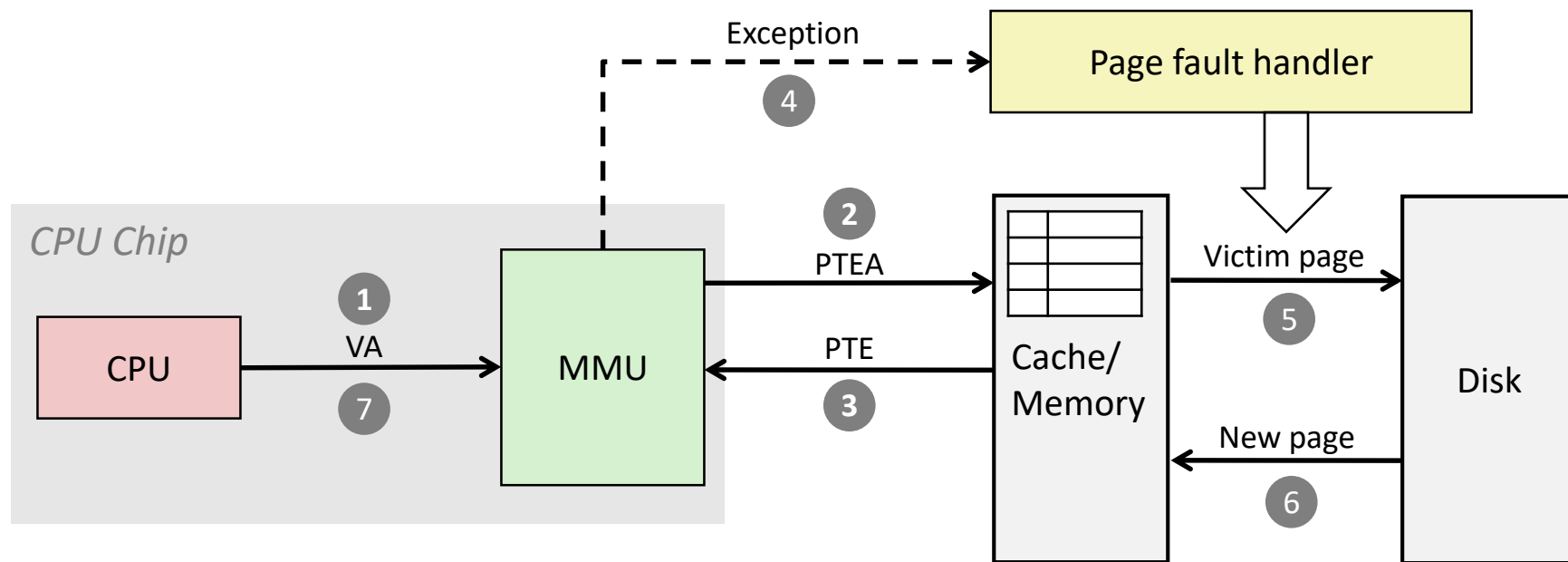
# Address Translation: Page Hit



- 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



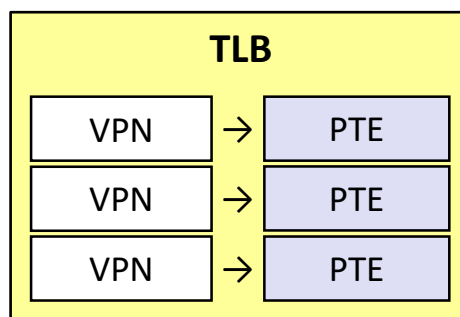
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

# Hmm... Translation Sounds Slow

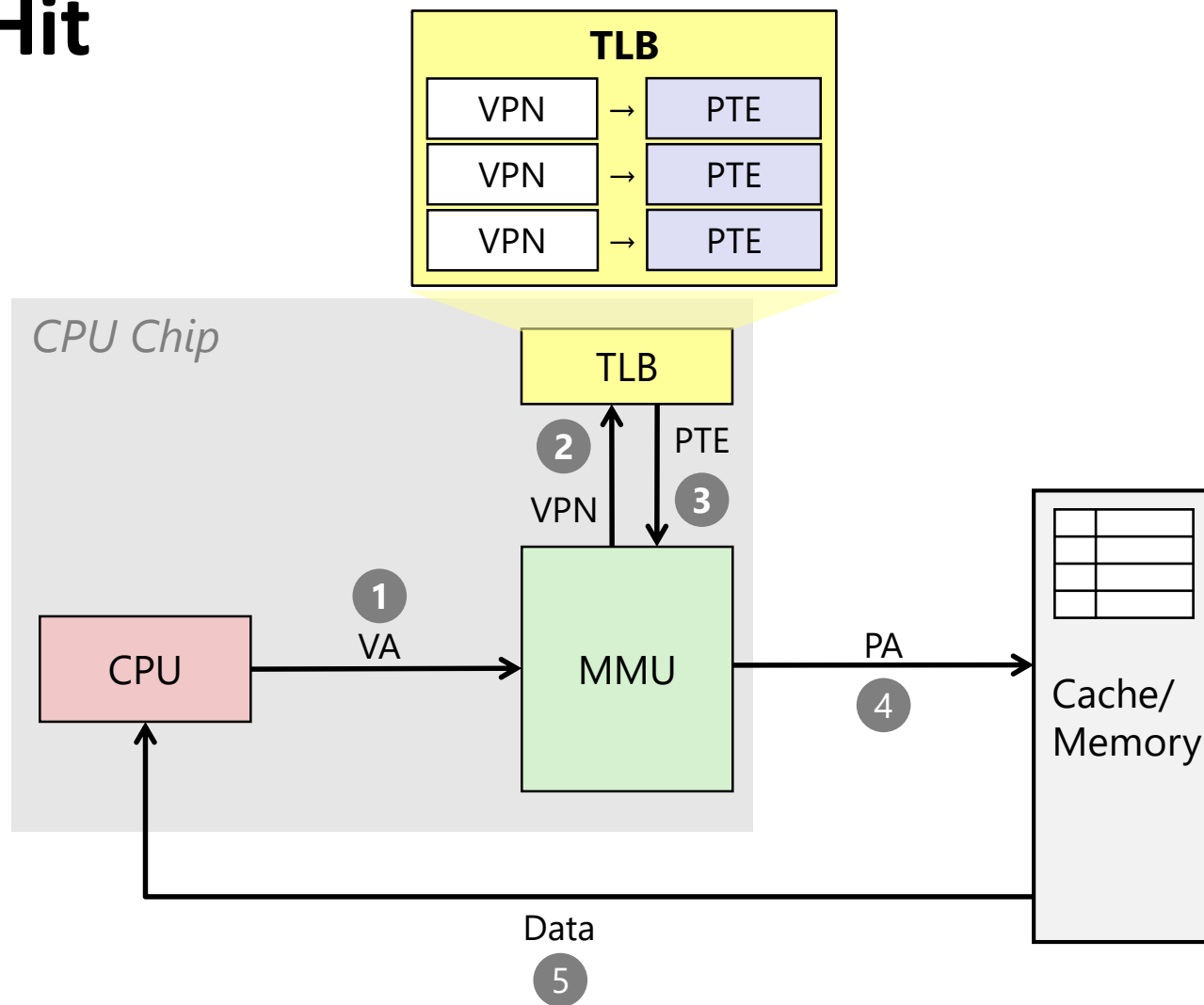
- ❖ The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles
- ❖ *What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

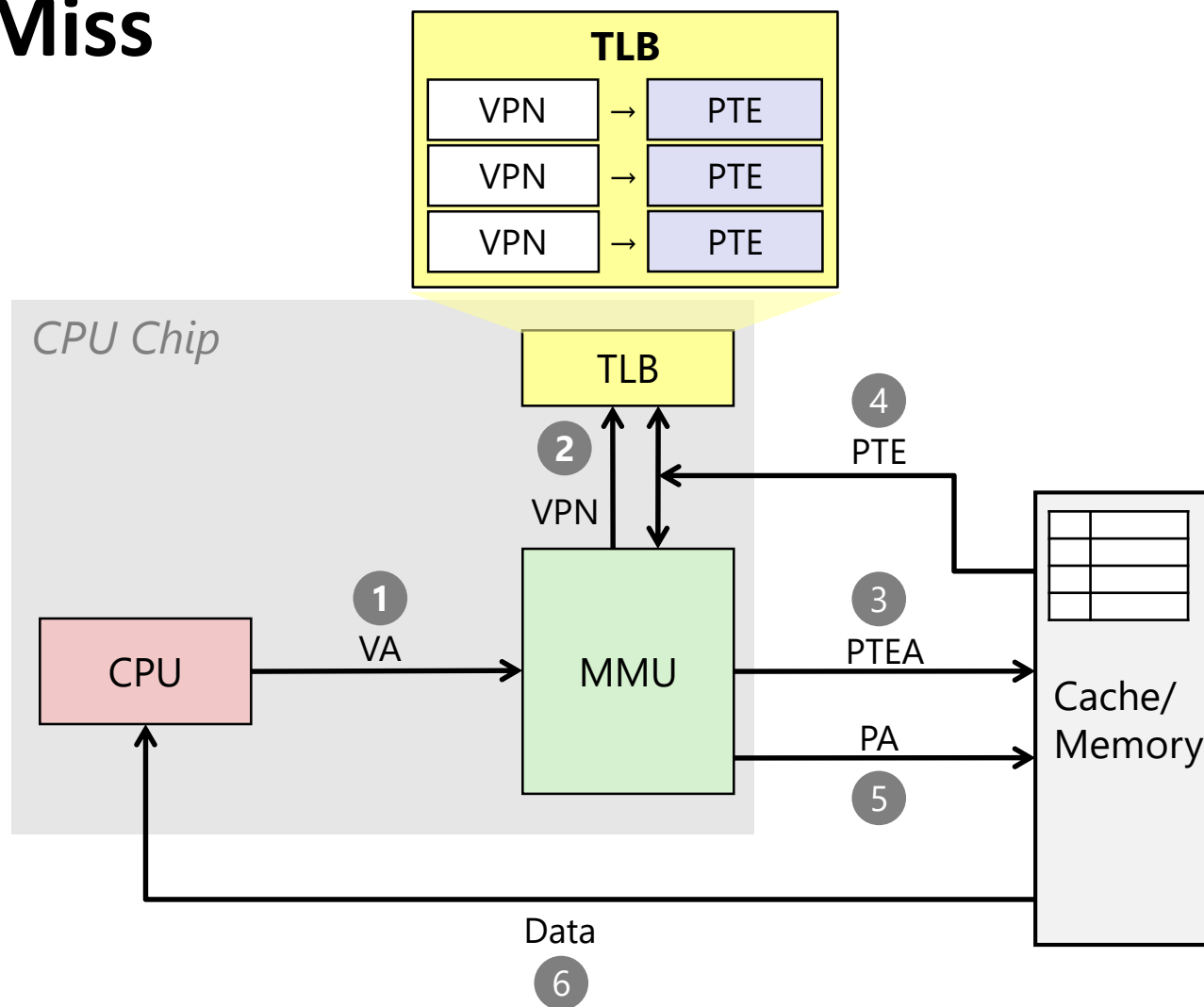


# TLB Hit



❖ A TLB hit eliminates a memory access!

# TLB Miss



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

# Fetching Data on a Memory Read

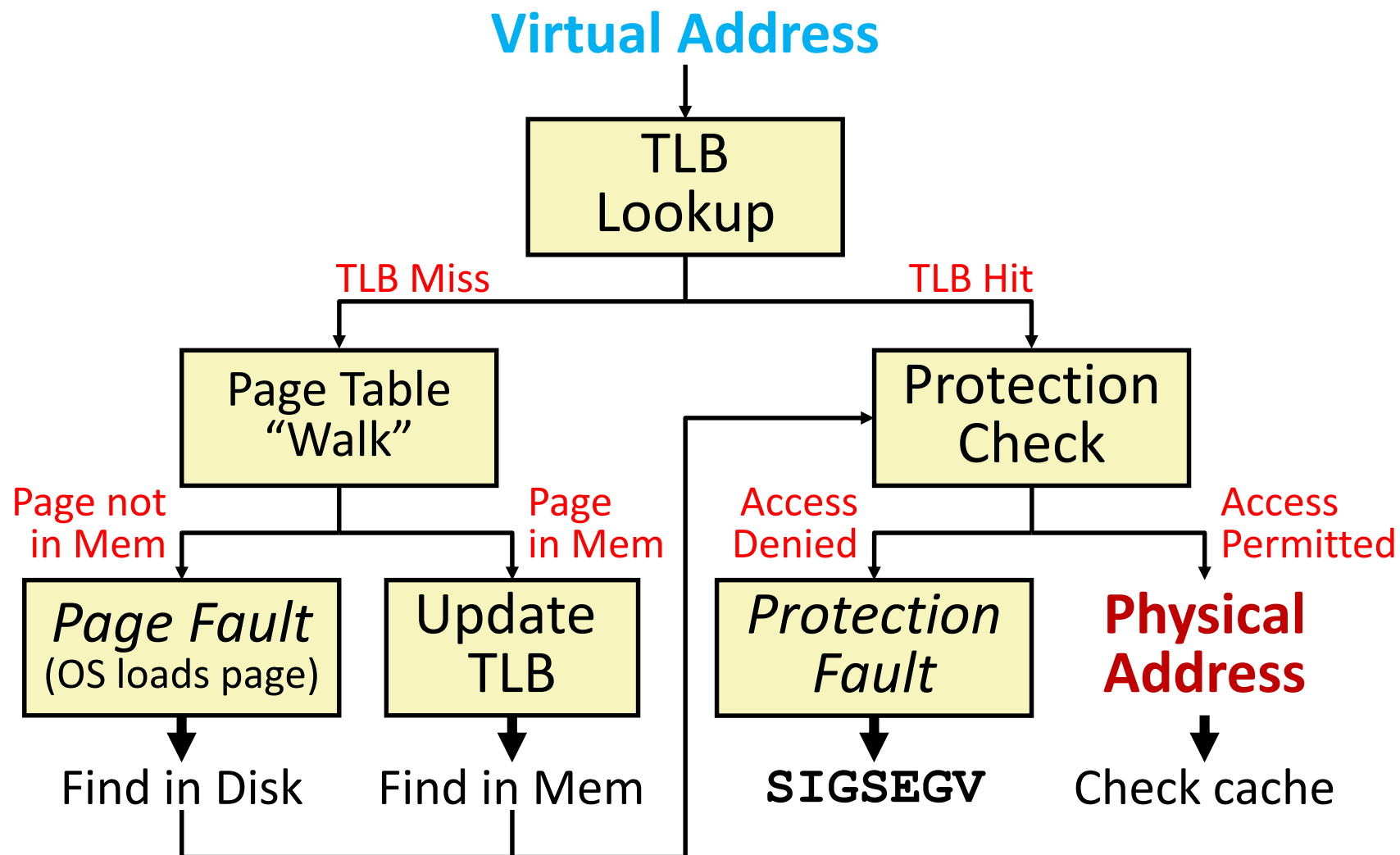
## 1) Check TLB

- 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





# 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

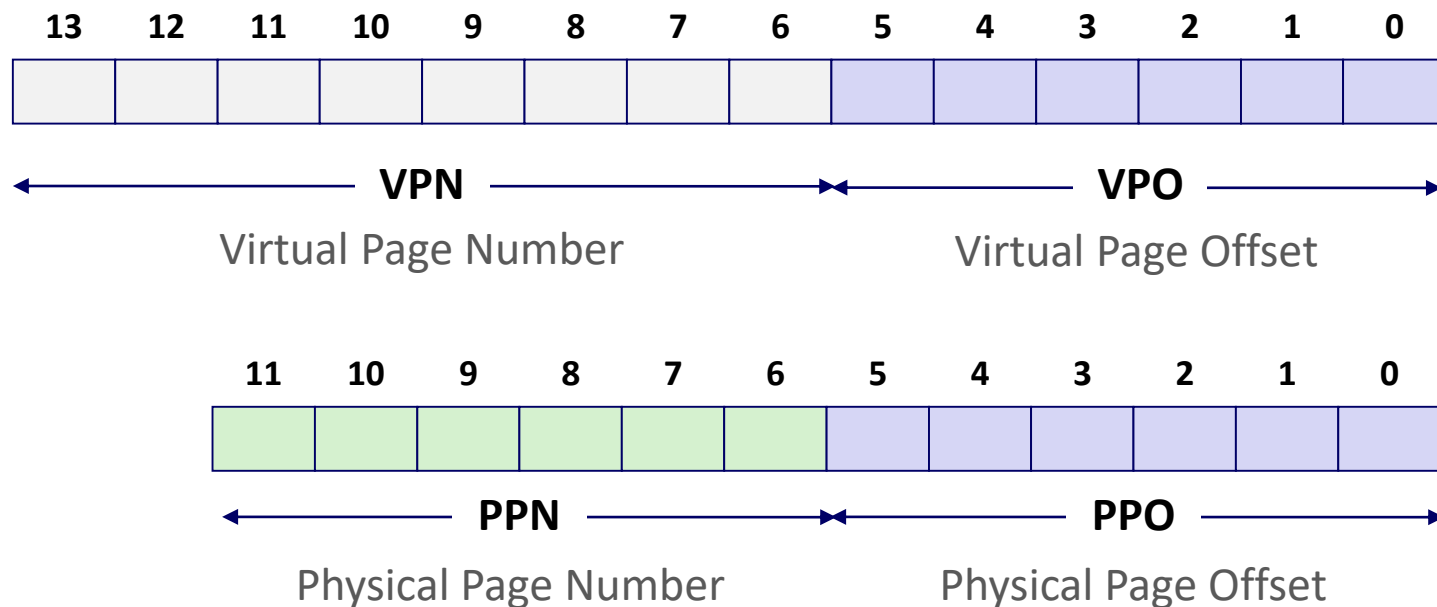
## ❖ Components of the physical address (PA)

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

# Simple Memory System Example (small)

## ❖ Addressing

- 14-bit virtual addresses
- 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

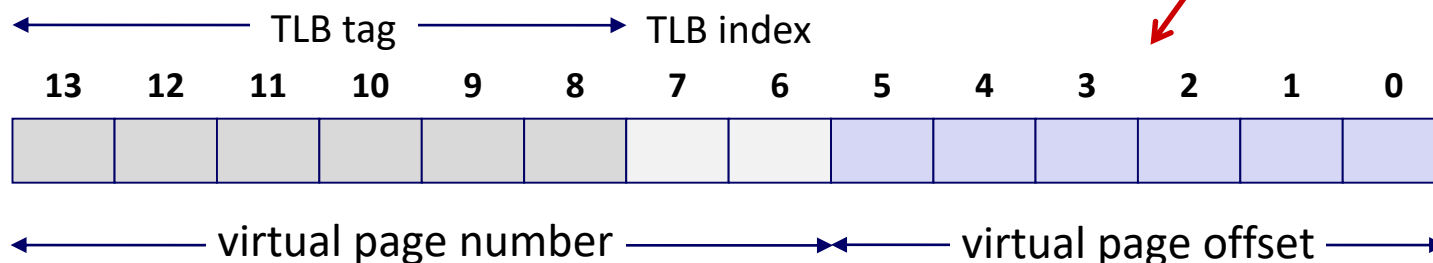
<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
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?

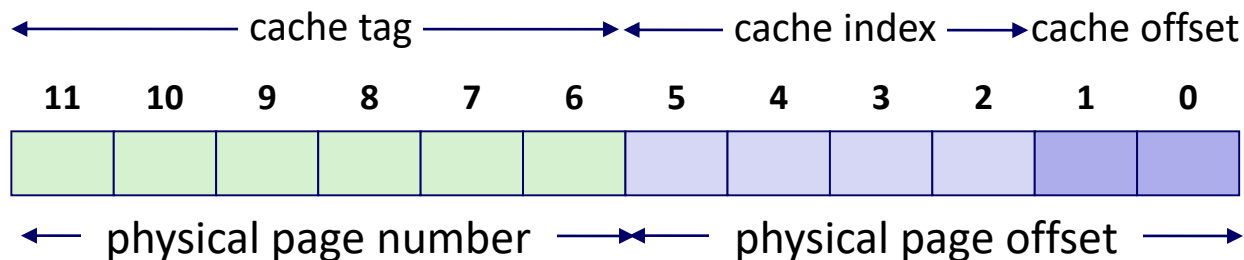


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

# Simple Memory System: Cache

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

- ❖ Direct-mapped with  $K = 4 \text{ B}$ ,  $C/K = 16$
- ❖ 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

## 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:

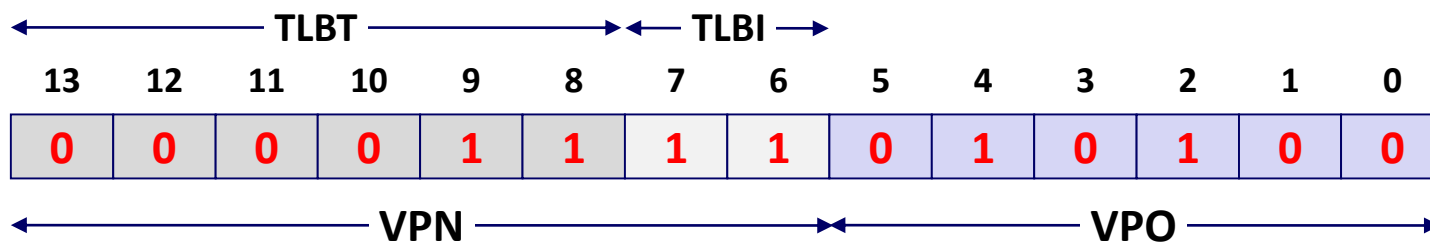
Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

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

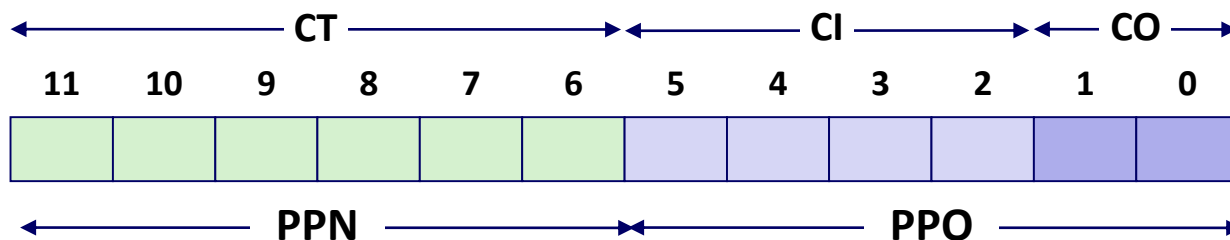
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x03D4



VPN \_\_\_\_\_ TLBT \_\_\_\_\_ TLBI \_\_\_\_\_ TLB Hit? \_\_\_\_ Page Fault? \_\_\_\_ PPN \_\_\_\_\_

❖ Physical Address:



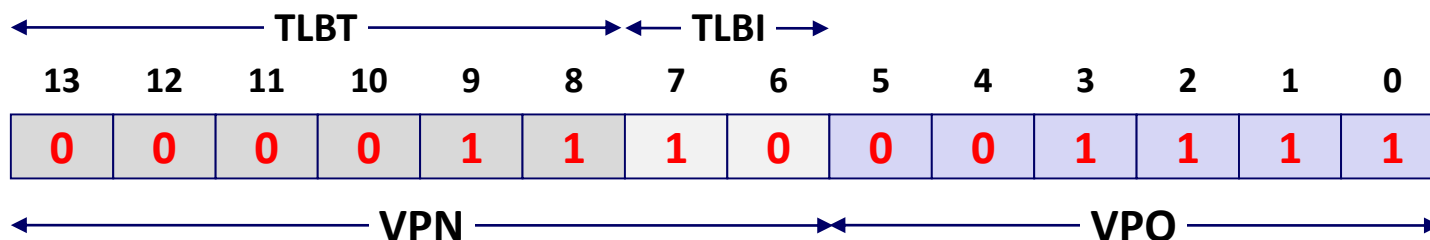
CT \_\_\_\_\_ CI \_\_\_\_\_ CO \_\_\_\_\_ Cache Hit? \_\_\_\_ Data (byte) \_\_\_\_\_



# Memory Request Example #2

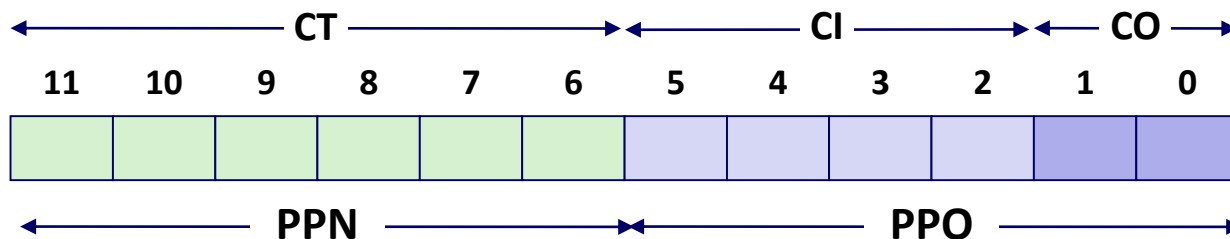
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x038F



VPN \_\_\_\_\_ TLBT \_\_\_\_\_ TLBI \_\_\_\_\_ TLB Hit? \_\_\_\_ Page Fault? \_\_\_\_ PPN \_\_\_\_\_

❖ Physical Address:

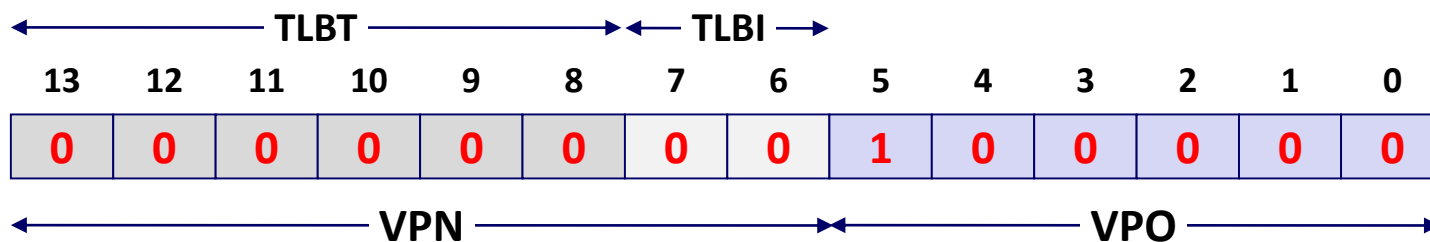


CT \_\_\_\_\_ CI \_\_\_\_\_ CO \_\_\_\_\_ Cache Hit? \_\_\_\_ Data (byte) \_\_\_\_\_

# Memory Request Example #3

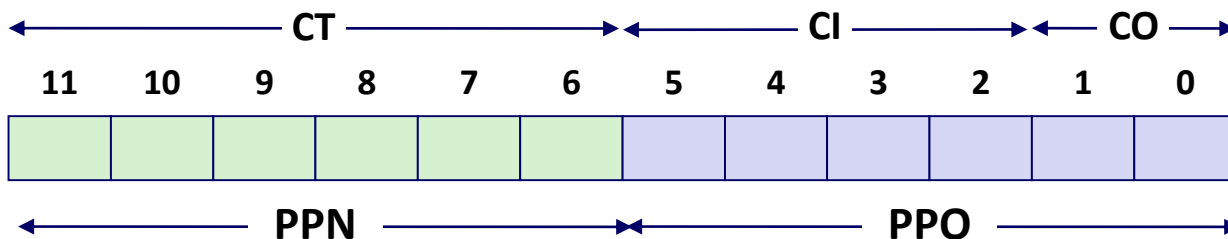
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x0020



VPN \_\_\_\_\_ TLBT \_\_\_\_\_ TLBI \_\_\_\_\_ TLB Hit? \_\_\_\_ Page Fault? \_\_\_\_ PPN \_\_\_\_\_

❖ Physical Address:

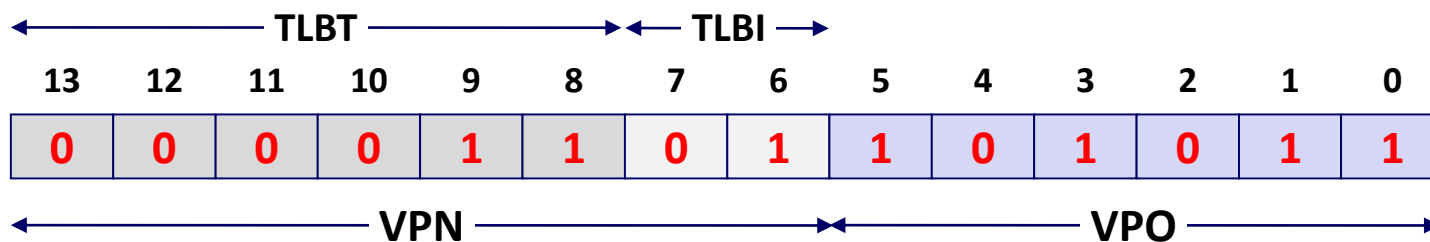


CT \_\_\_\_\_ CI \_\_\_\_\_ CO \_\_\_\_\_ Cache Hit? \_\_\_\_ Data (byte) \_\_\_\_\_

# Memory Request Example #4

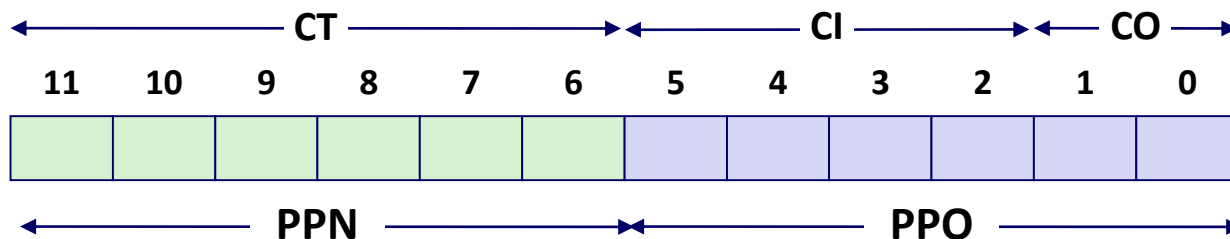
**Note:** It is just coincidence that the PPN is the same width as the cache Tag

❖ Virtual Address: 0x036B



VPN \_\_\_\_\_ TLBT \_\_\_\_\_ TLBI \_\_\_\_\_ TLB Hit? \_\_\_\_ Page Fault? \_\_\_\_ PPN \_\_\_\_\_

❖ Physical Address:



CT \_\_\_\_\_ CI \_\_\_\_\_ CO \_\_\_\_\_ Cache Hit? \_\_\_\_ Data (byte) \_\_\_\_\_

# 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)