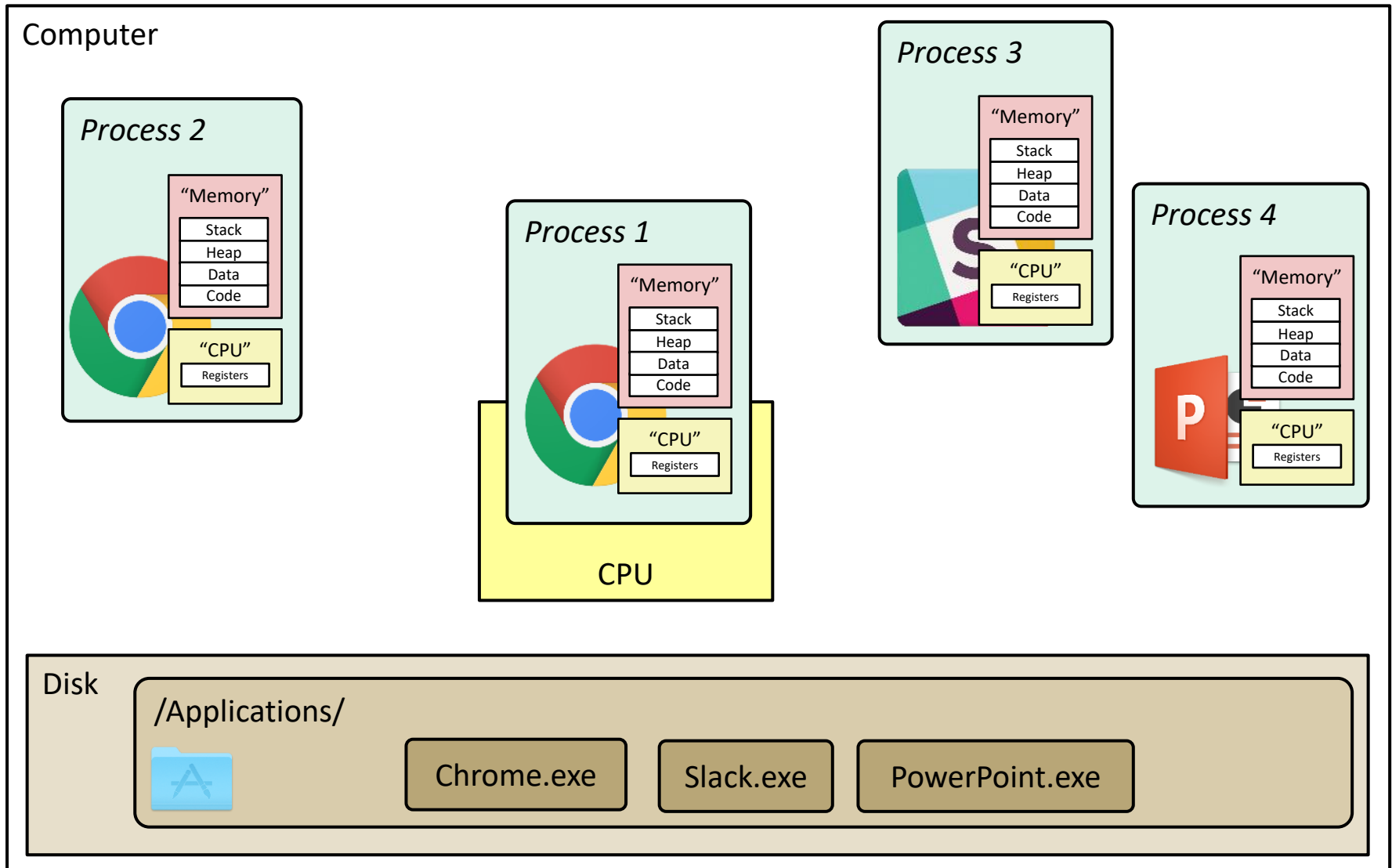# Caches and Virtual Memory

CMPT 295 Week 8

Processes

❖ A ***process*** is an instance of a running program

 ■ Provided by the OS

  • OS uses a data structure (PCB) to represent each process

 ■ Maintains the ***interface*** between the program and the underlying hardware (CPU + memory)

❖ Process provides each program with two key abstractions:

 ■ *Logical control flow*

  • Each program seems to have exclusive use of the CPU

  • Provided by kernel mechanism called ***context switching***

 ■ *Private address space*

  • Each program seems to have exclusive use of main memory

  • Provided by kernel mechanism called ***virtual memory***

Memory
Stack
Heap
Data
Code

CPU
Registers
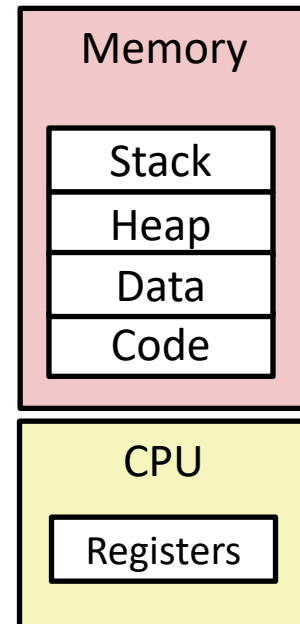
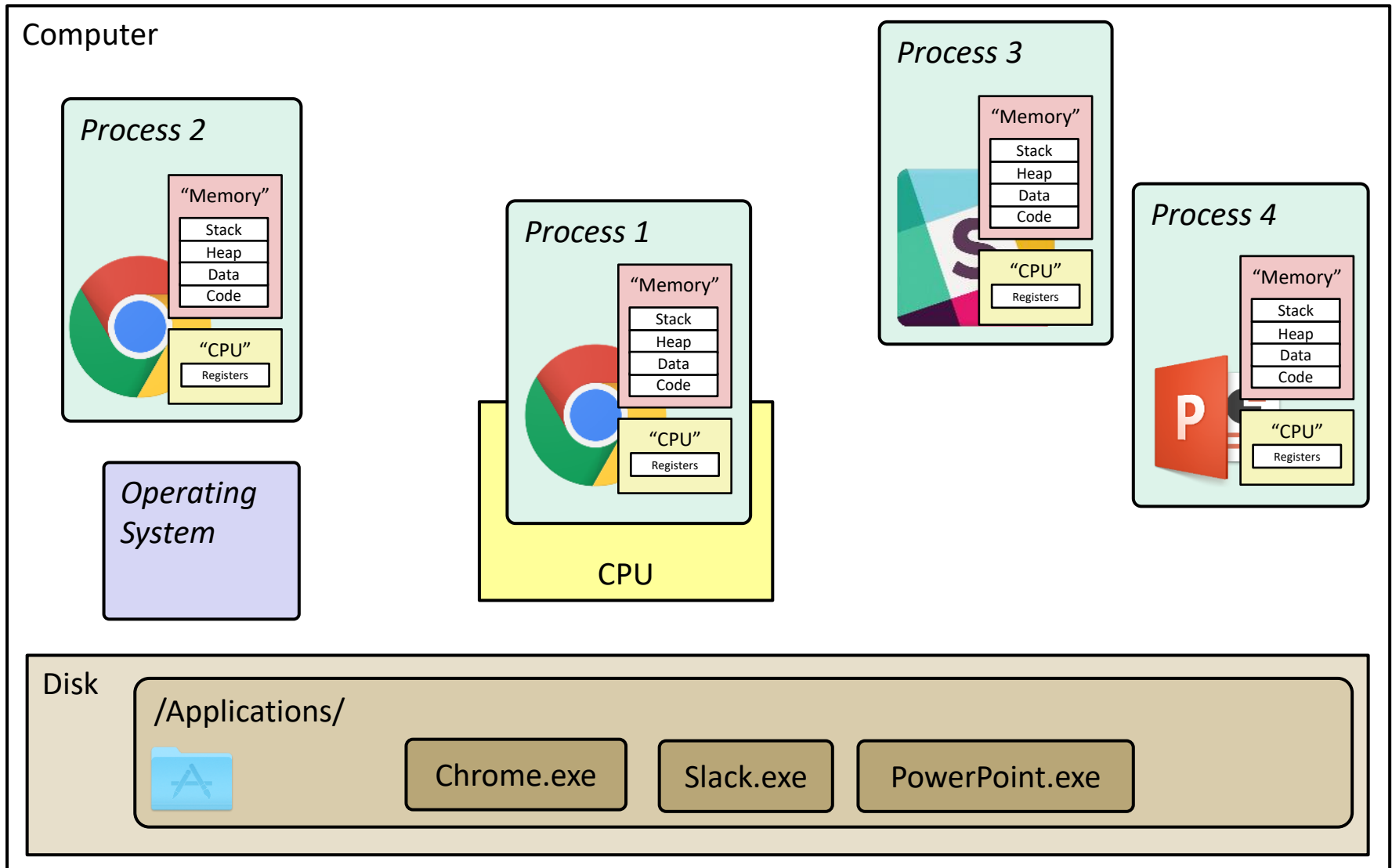# What is a process?   It's an *illusion*!

What is a process?   It's an *illusion*!

Multiprocessing: The Illusion

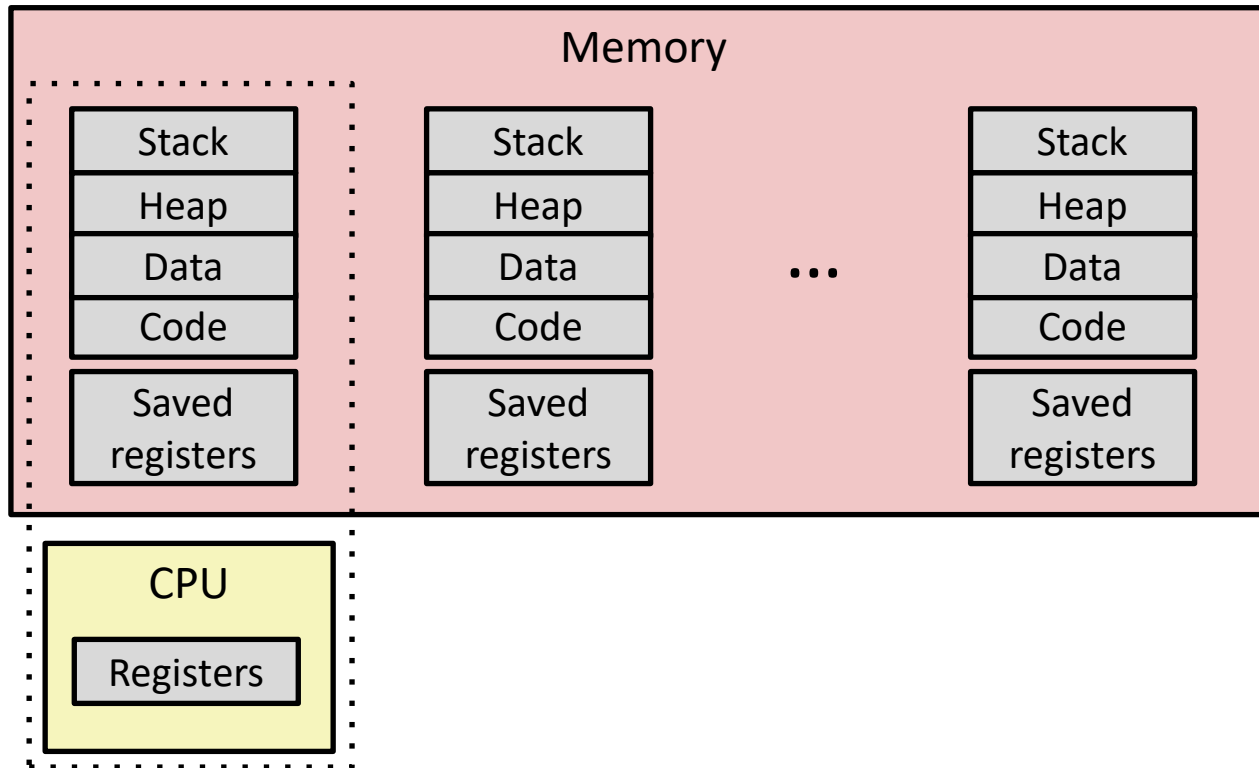| Memory		Memory			Memory
Stack		Stack			Stack
Heap		Heap	**. . .**		Heap
Data		Data			Data
Code		Code			Code
CPU		CPU			CPU
Registers		Registers			Registers
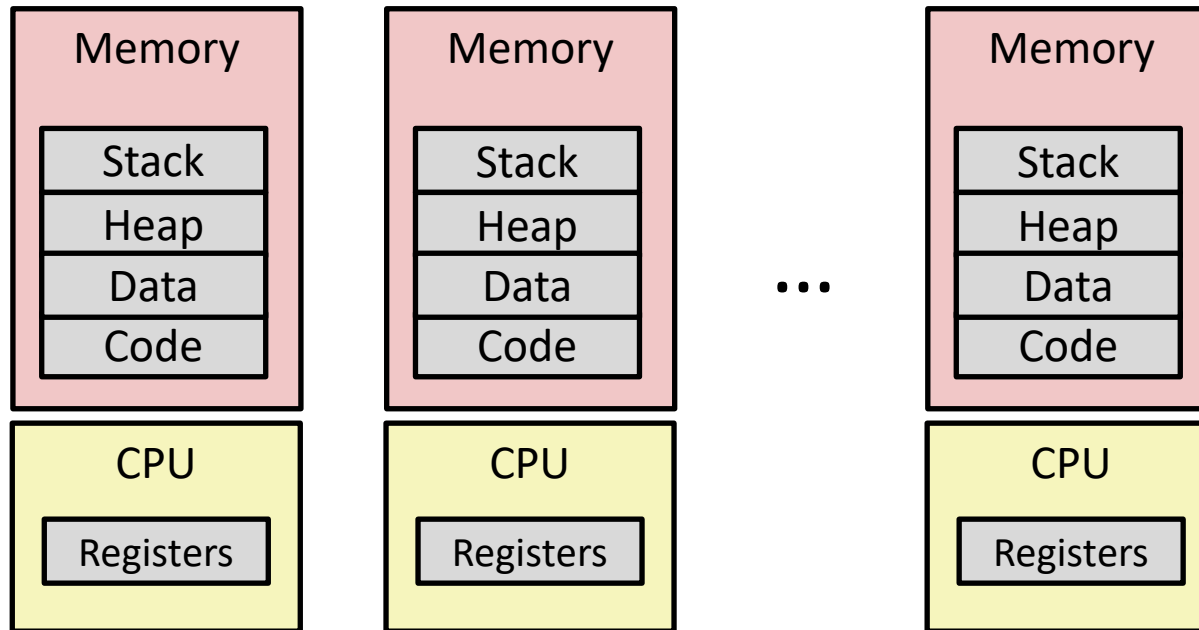
❖ Computer runs many processes simultaneously

- Applications for one or more users
  - Web browsers, email clients, editors, …

- Background tasks
  - Monitoring network & I/O devices

Multiprocessing: The Reality

❖ Single processor executes multiple processes *concurrently*

- Process executions interleaved, CPU runs *one at a time*
- Address spaces managed by virtual memory system (today's lecture)
- *Execution context* (register values, stack, …) for other processes saved in memory

6

Multiprocessing

❖ Context switch

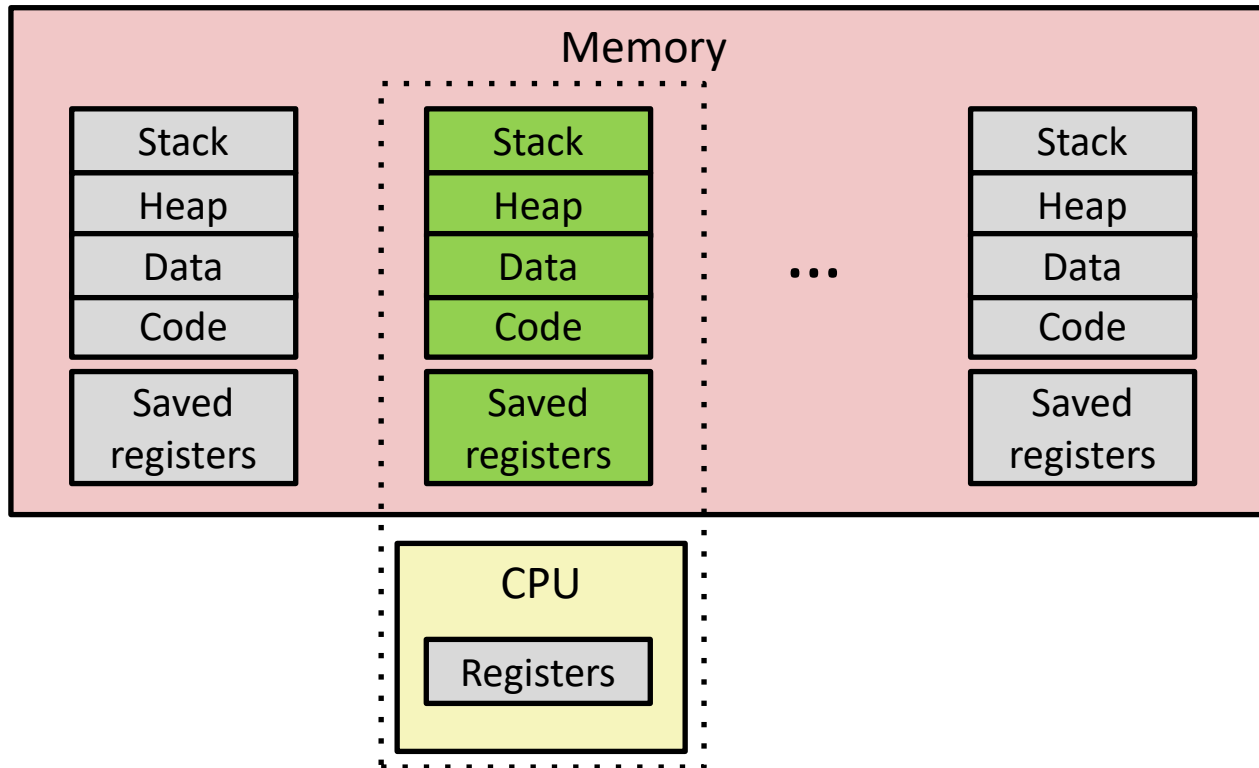1)   **Save current registers in memory**

Multiprocessing

❖ Context switch

1) Save current registers in memory

2) **Schedule next process for execution**

Multiprocessing
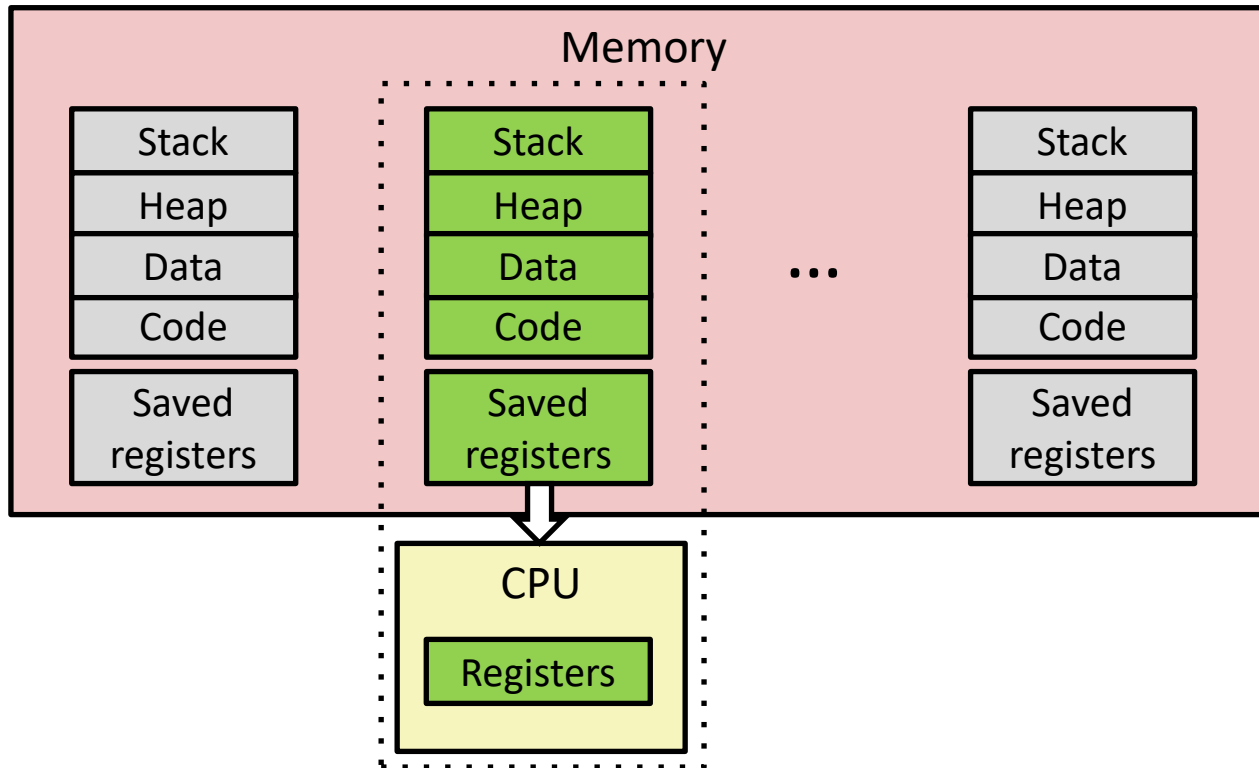
❖ Context switch

1) Save current registers in memory

2) Schedule next process for execution

3) **Load saved registers and switch address space**

Multiprocessing on Multicore Processors

Memory
Stack · Heap · Data · Code · Saved registers · · · Stack · Heap · Data · Code · Saved registers

CPU — Registers · CPU — Registers
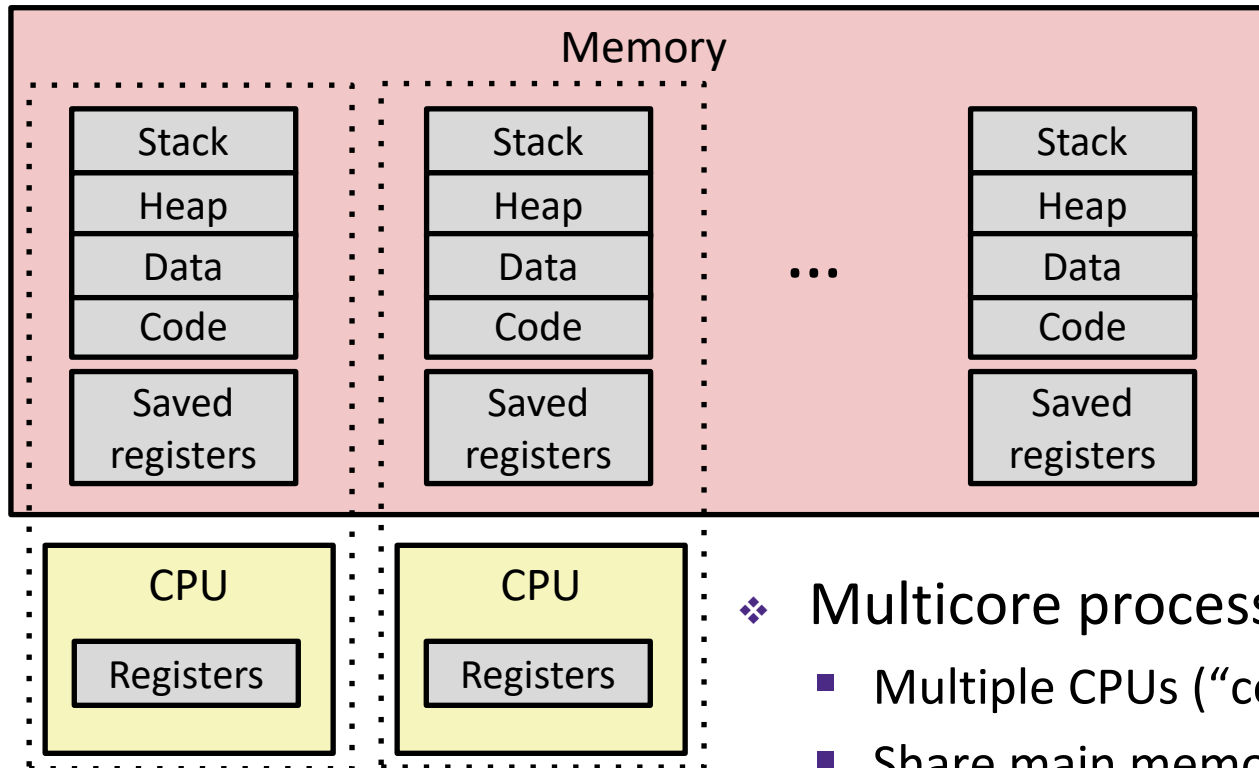
❖ Multicore processors

▪ Multiple CPUs ("cores") on single chip

▪ Share main memory (and some of the caches)

▪ Each can execute a separate process

• Kernel schedules processes to cores

• *Still* **constantly swapping processes**

Memory as we know it so far… is *virtual!*

❖ Programs refer to virtual memory addresses   0xFF······F

- Conceptually memory is just a very large array of bytes
- System provides private address space to each process

❖ Allocation: Compiler and run-time system

- Where different program objects should be stored
- All allocation within single virtual address space

❖ But…

- We *probably* don't have 2^w bytes of physical memory
- We *certainly* don't have 2^w bytes of physical memory *for every process*
- Processes should not interfere with one another
  - Except in certain cases where they want to share code or data 0x00······0

11
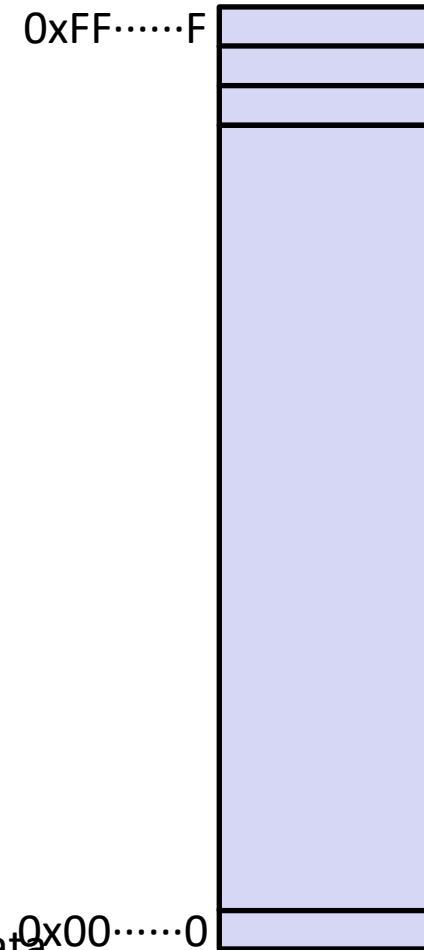
Why Virtual Memory (VM)?

❖ Efficient use of limited main memory (RAM)

- ■ Use RAM as a cache for the parts of a virtual address space
  - • Some non-cached parts stored on disk
  - • Some (unallocated) non-cached parts stored nowhere
- ■ Keep only active areas of virtual address space in memory
  - • Transfer data back and forth as needed

❖ Simplifies memory management for programmers

- ■ Each process "gets" the same full, private linear address space

❖ Isolates address spaces (protection)

- ■ One process can't interfere with another's memory
  - • They operate in *different address spaces*
- ■ User process cannot access privileged information
  - • Different sections of address spaces have different permissions

12

A System Using Virtual Addressing

Main memory

CPU Chip

CPU → Virtual address (VA) `0x4100` → MMU → Physical address (PA) `0x4`

0:
1:
2:
3:
4:
5:
6:
7:
8:
...
M-1:

Memory Management Unit

Data (int/float)
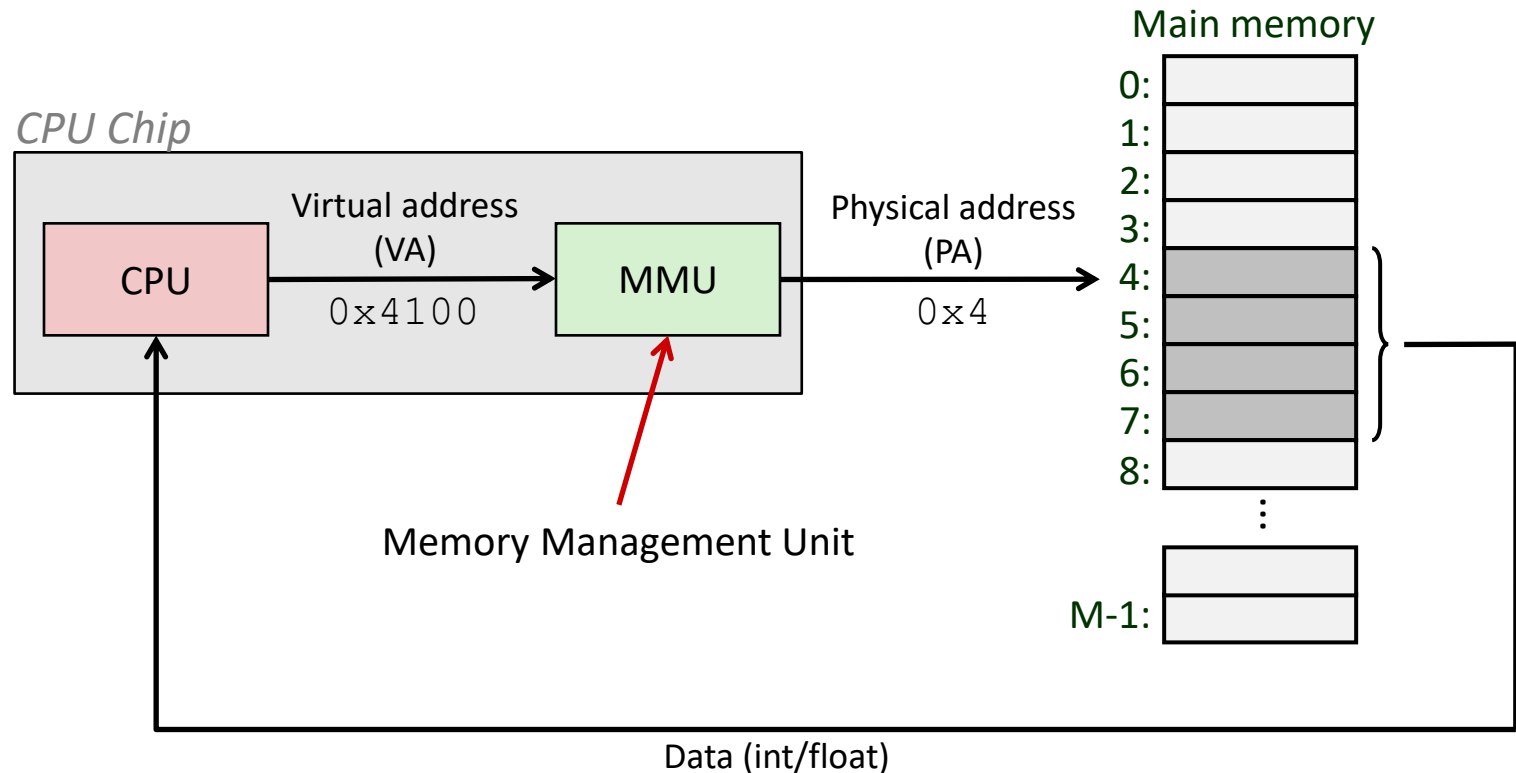
- ❖ Physical addresses are *completely invisible to programs*
  - ■ Used in all modern desktops, laptops, servers, smartphones…
  - ■ One of the great ideas in computer science

13

Indirection and Address Spaces

Virtual memory

Process 1

Virtual memory

Process n

mapping
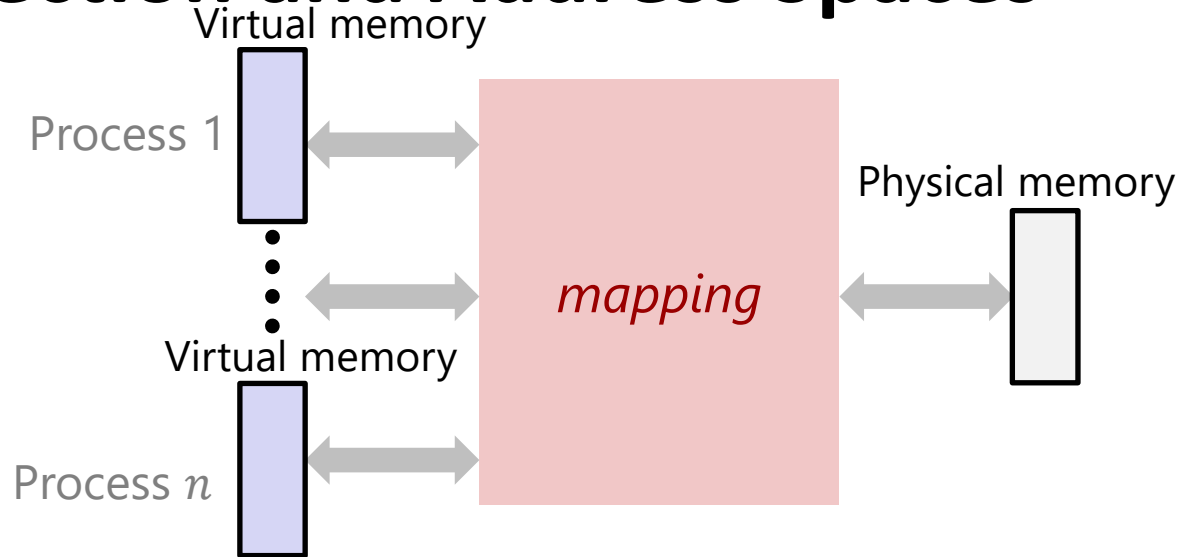
Physical memory

❖ Virtual address space: Set of $N = 2^n$ virtual addr

   ▪ {0, 1, 2, 3, …, N-1}

❖ Physical address space: Set of $M = 2^m$ physical addr

   ▪ {0, 1, 2, 3, …, M-1}

❖ Every byte in main memory has:

   ▪ one physical address (PA)

   ▪ zero, one, *or more* virtual addresses (VAs)

Memory Overview

❖ `LD t1, 0 (sp)`

requested 32-bits

CPU

MMU

TLB

Cache

Line

Main memory
(DRAM)

Page

Block

Disk
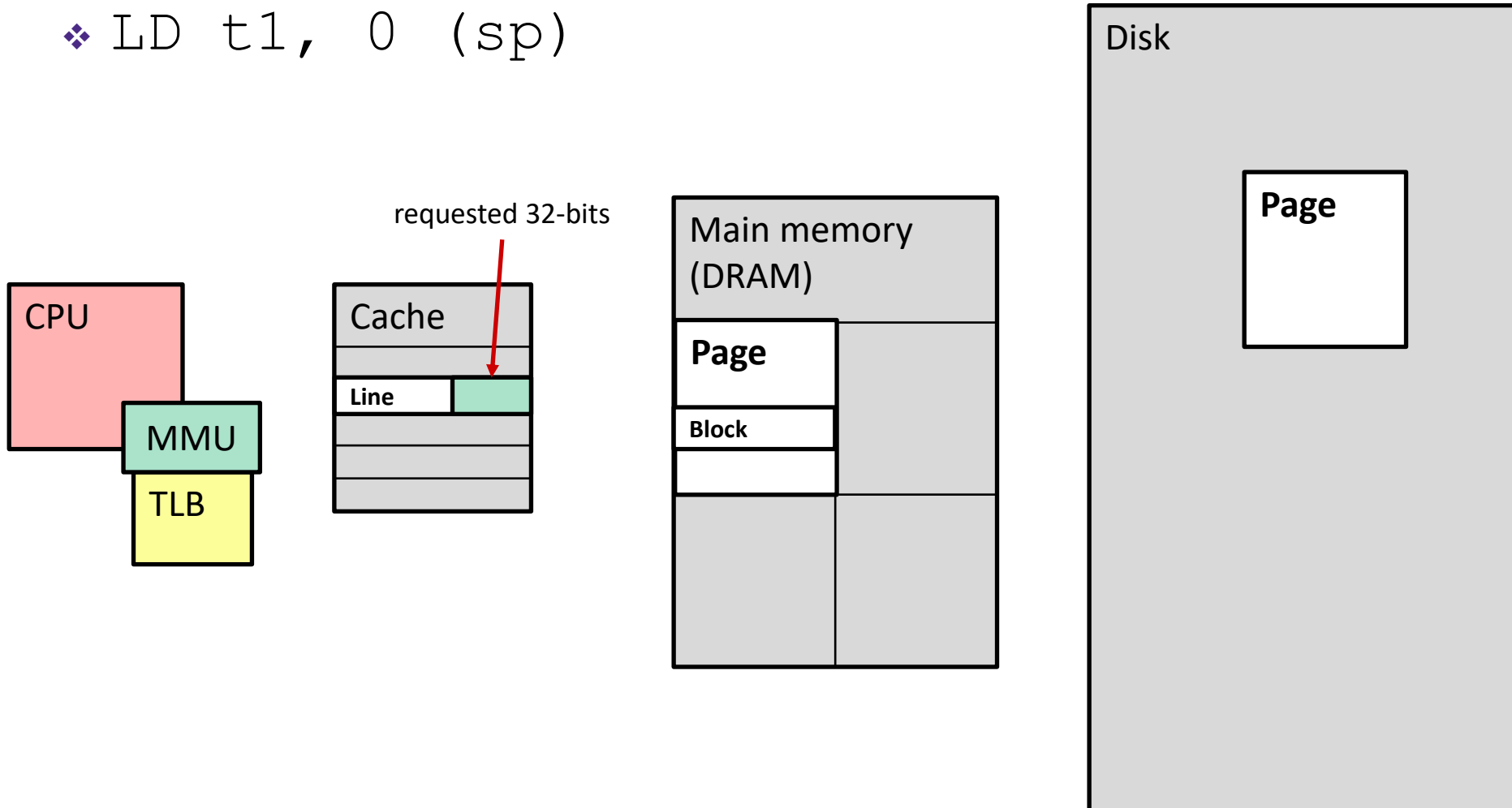
Page

Address Translation

> ### *How do we perform the virtual*
> ### *→ physical address translation?*

CPU Chip

CPU

Virtual address
(VA)
`0x4100`

MMU

Physical address
(PA)
`0x4`

Memory Management Unit

Data (int/float)

Main memory

0:
1:
2:
3:
4:
5:
6:
7:
8:
⋮
M-1:
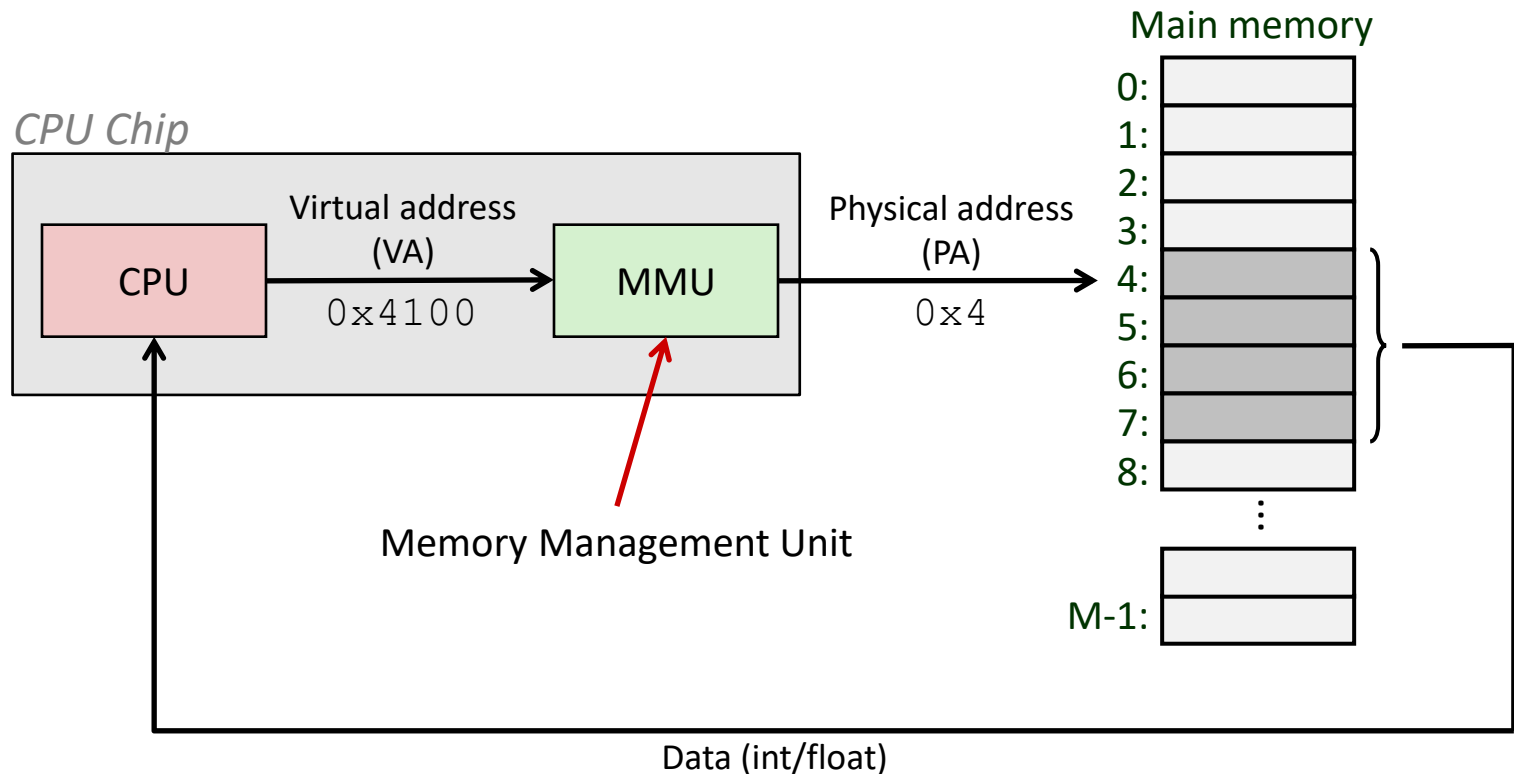
Address Translation: Page Tables

❖ CPU-generated address can be split into:

n-bit address: | Virtual Page Number | Page Offset |

- Request is Virtual Address (VA), want Physical Address (PA)
- Note that Physical Offset = Virtual Offset (page-aligned)

❖ Use lookup table that we call the *page table* (PT)

- Replace Virtual Page Number (VPN) for Physical Page Number (PPN) to generate Physical Address
- Index PT using VPN: page table entry (PTE) stores the PPN plus management bits (*e.g.* Valid, Dirty, access rights)
- Has an entry for *every* virtual page

Address Translation: Page Hit
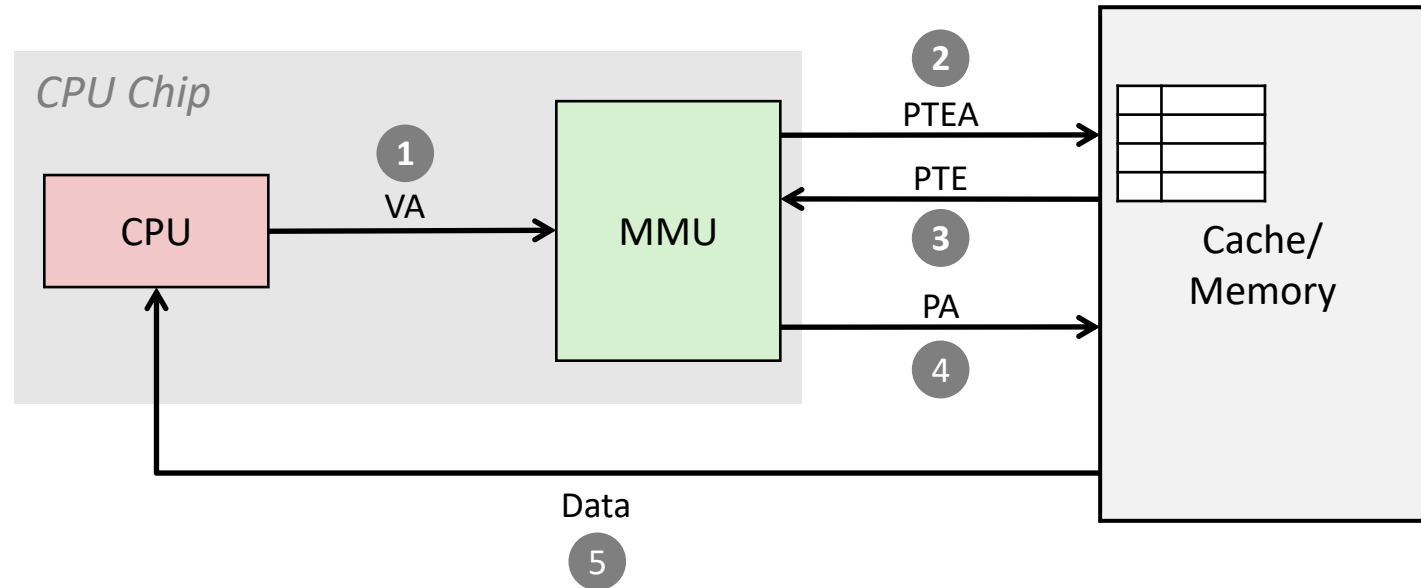
1) Processor sends *virtual* address to MMU (*memory management unit*)

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
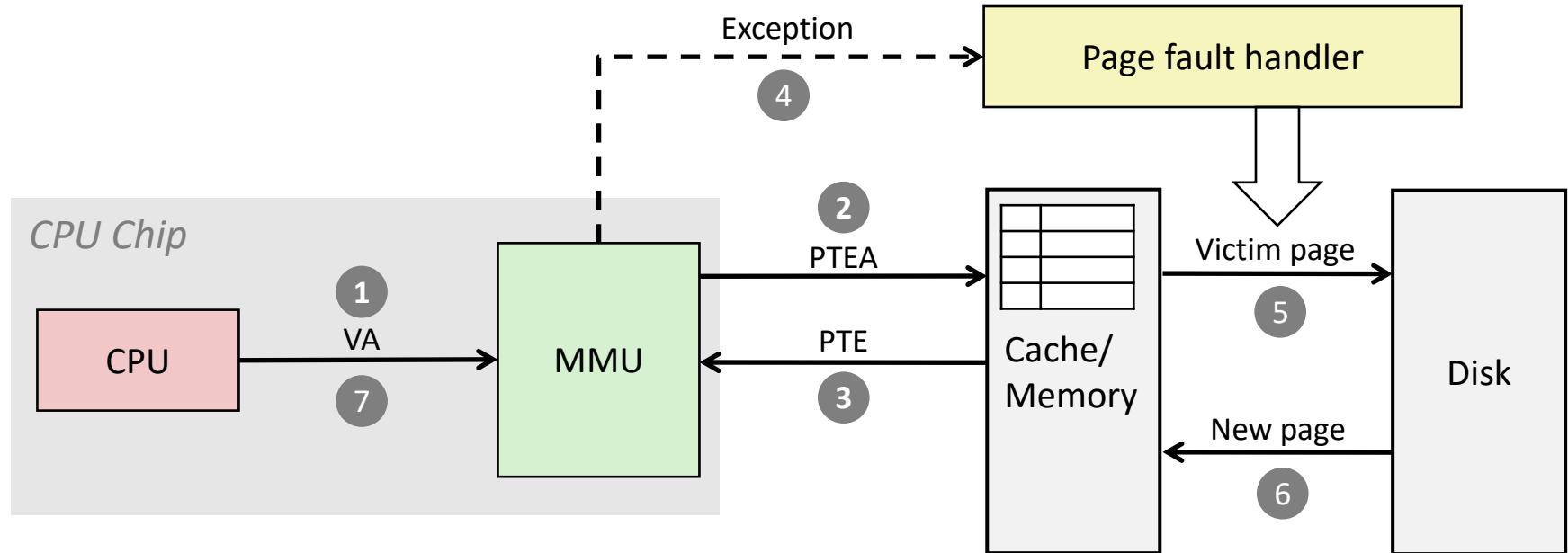
1) Processor sends virtual address to MMU

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

Page Hit

❖ *Page hit:* VM reference is in physical memory

Page Table (DRAM)

Physical memory (DRAM)

Virtual address

Valid	PPN/Disk Addr
PTE 0 0	null
1	●
1	●
0	●
1	●
0	null
0	●
PTE 7 1	●
...	...

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Virtual memory (DRAM/disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7
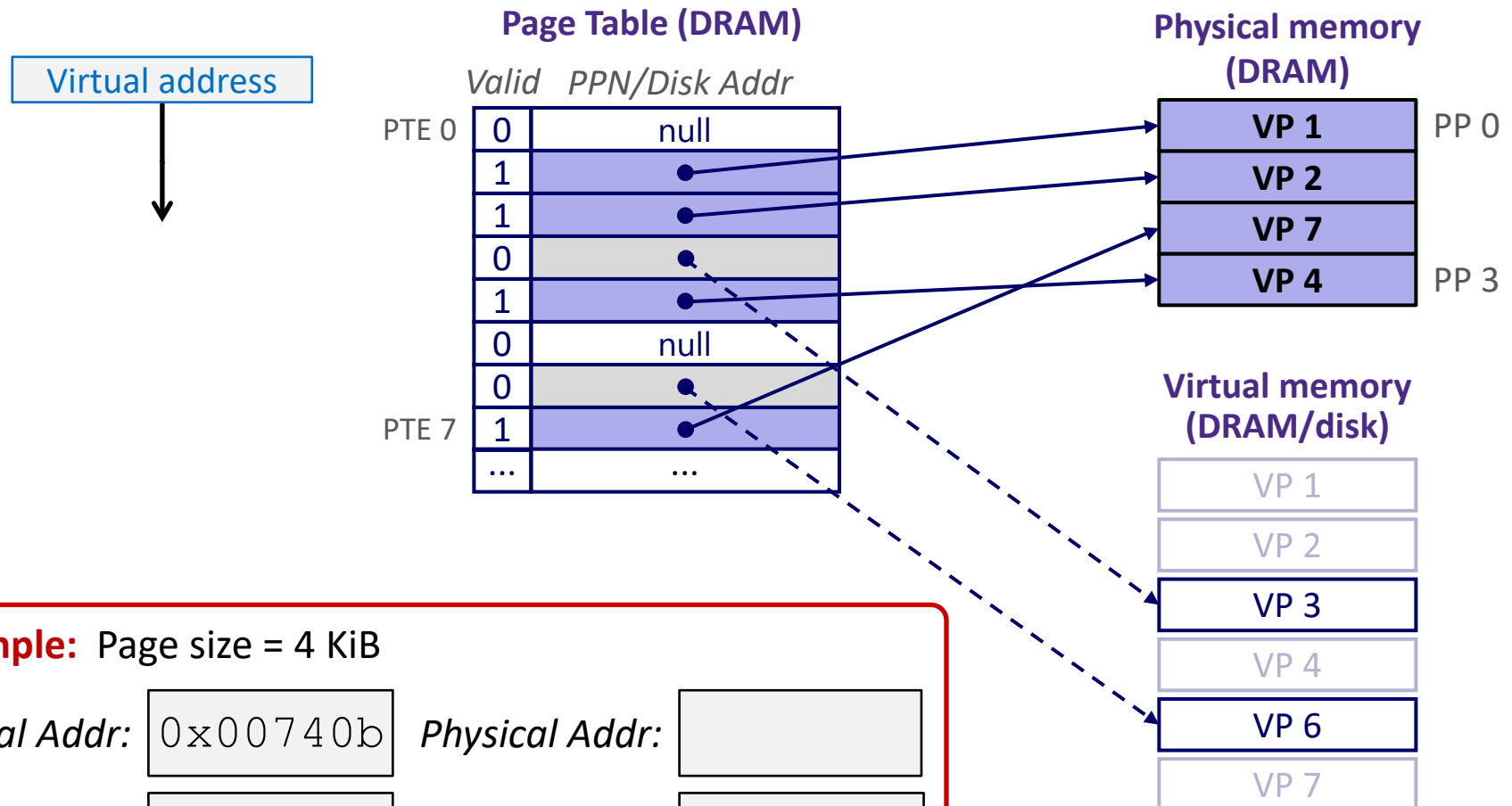
Example: Page size = 4 KiB

Virtual Addr: `0x00740b`   *Physical Addr:*

VPN: *PPN:*

Page Fault

❖ *Page fault:* VM reference is NOT in physical memory

Page Table (DRAM)

Physical memory (DRAM)

Virtual address

Valid   *PPN/Disk Addr*

	Valid	PPN/Disk Addr
PTE 0	0	null
	1	●
	1	●
	0	●
	1	●
	0	null
	0	●
PTE 7	1	●
	...	...

VP 1	PP 0
VP 2	
VP 7	
VP 4	PP 3

Virtual memory (DRAM/disk)

VP 1
VP 2
VP 3
VP 4
VP 6
VP 7
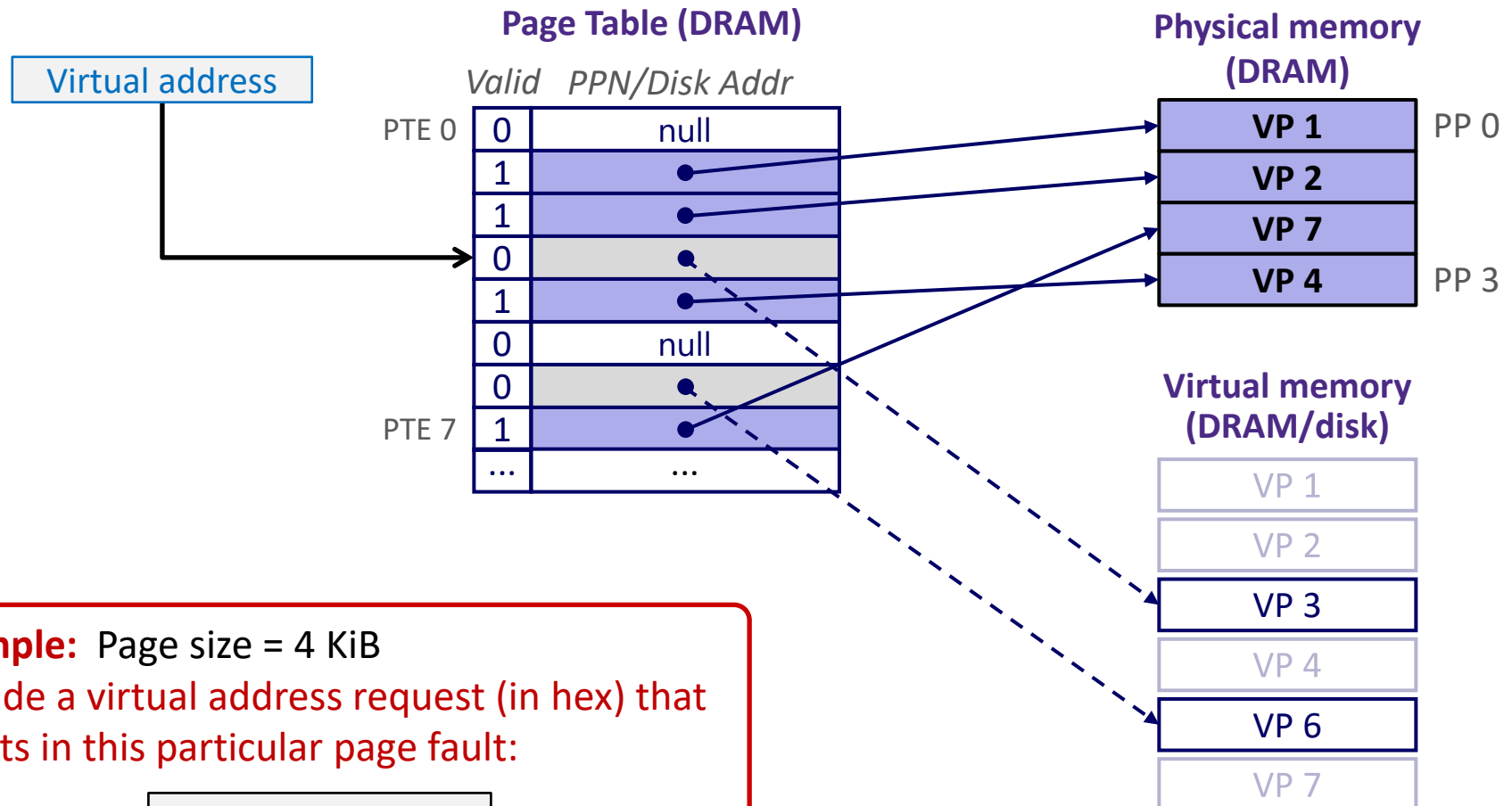
Example: Page size = 4 KiB
Provide a virtual address request (in hex) that results in this particular page fault:

Virtual Addr:

21

Address Translation Is 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 3-4 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

  - 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

- Much faster than a page table lookup in cache/memory
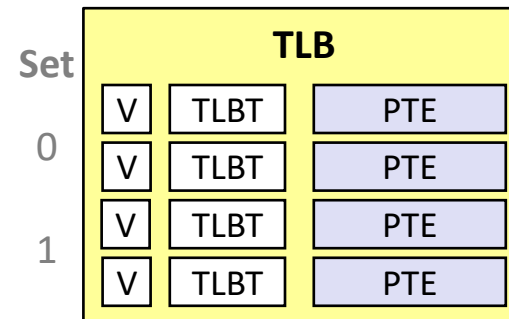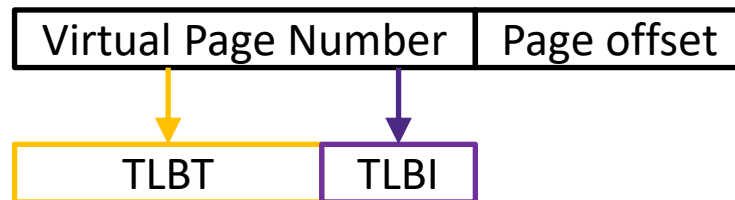
TLB Hit
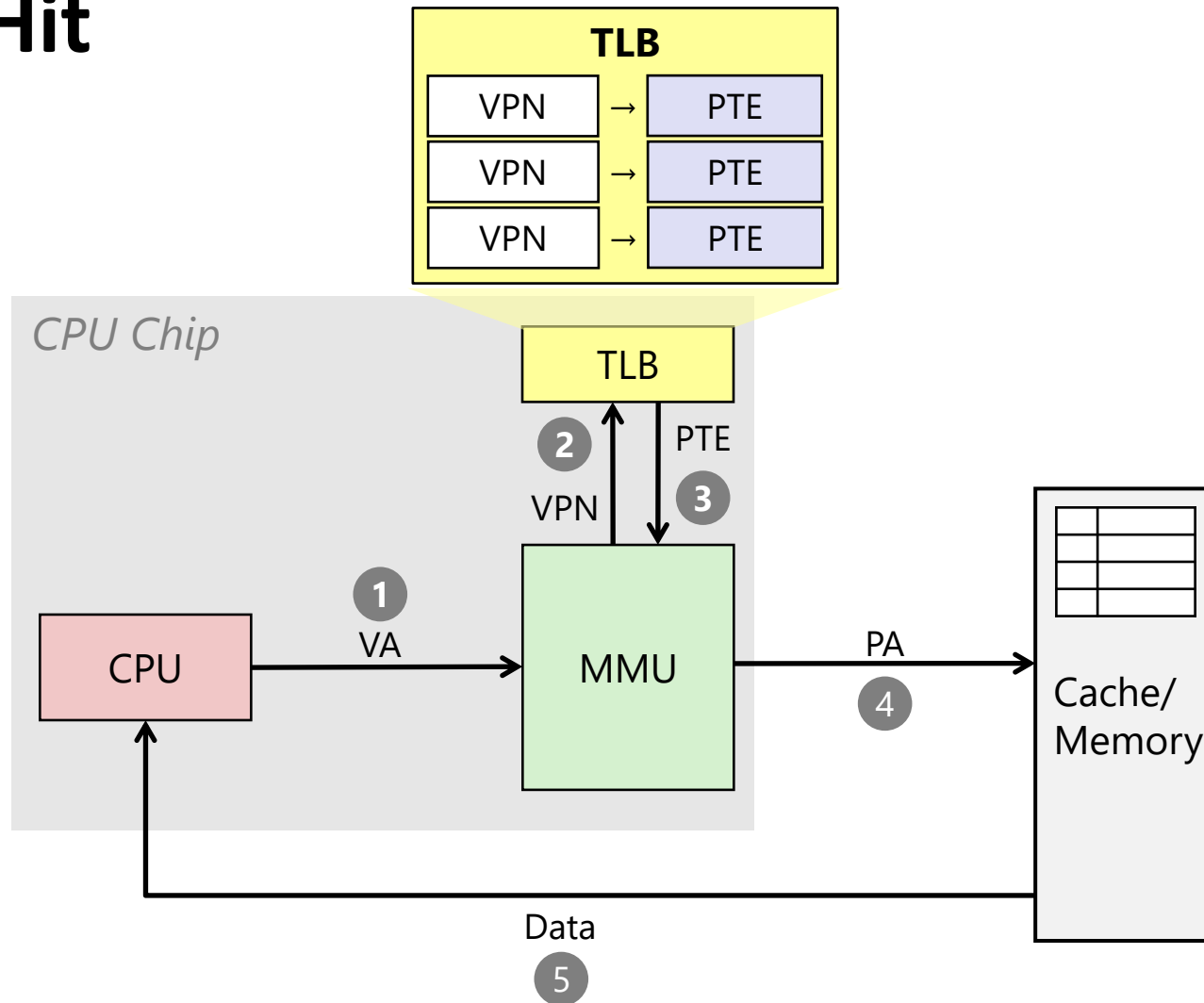
❖ A TLB hit eliminates a memory access!

Address Translation

Virtual Address

```
TLB Lookup
```

TLB Miss   *TLB Hit*

```
Check the
Page Table
```
```
Protection
Check
```

Page not in Mem   *Page in Mem*   *Access Denied*   *Access Permitted*

```
Page Fault
(OS loads page)
```
```
Update
TLB
```
```
Protection
Fault
```
Physical Address

Find in Disk   Find in Mem   `SIGSEGV`   Check cache

Miss   *Hit*

Address Manipulation

request from CPU:

n-bit virtual address

split to access TLB:

TLB Tag	TLB Index	Page Offset

(on TLB miss) access PT:

Virtual Page Number	Page offset

TRANSLATION

m-bit physical address:

Physical Page Number	Page offset

split to access cache:

Cache Tag	Cache Index	Offset

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

Peer Question

❖ How many bits wide are the following fields?

- 16 KiB pages

- 48-bit virtual addresses

- 16 GiB physical memory

	VPN	PPN
(A)	34	24
(B)	32	18
(C)	30	20
(D)	34	20

Some VM Topics Not Covered in 295

❖ Supporting more than one page size

  ▪ Small pages lead to more TLB misses; large pages lead to fragmentation

  ▪ Modern CPUs support >1 page size (e.g., 4KB, 2MB, 1GB)

❖ Multi-level page tables

  ▪ Needed to support large VM address space

  ▪ https://en.wikipedia.org/wiki/Intel_5-level_paging

❖ TLB hierarchy

  ▪ Modern CPUs have more than one level in TLB

  ▪ L1 usually split into two structures, one for instructions (iTLB) and another for data (dTLB). Entries 32-256

  ▪ L2 TLB unified (instructions and data), 512-2048 entries

Simple Memory System Example (small)

❖ Addressing

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

13	12	11	10	9	8	7	6	5	4	3	2	1	0

←————————— **VPN** —————————→←————————— **VPO** —————————→

Virtual Page Number   Virtual Page Offset

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

←————————— **PPN** —————————→←————————— **PPO** —————————→

Physical Page Number   Physical Page Offset
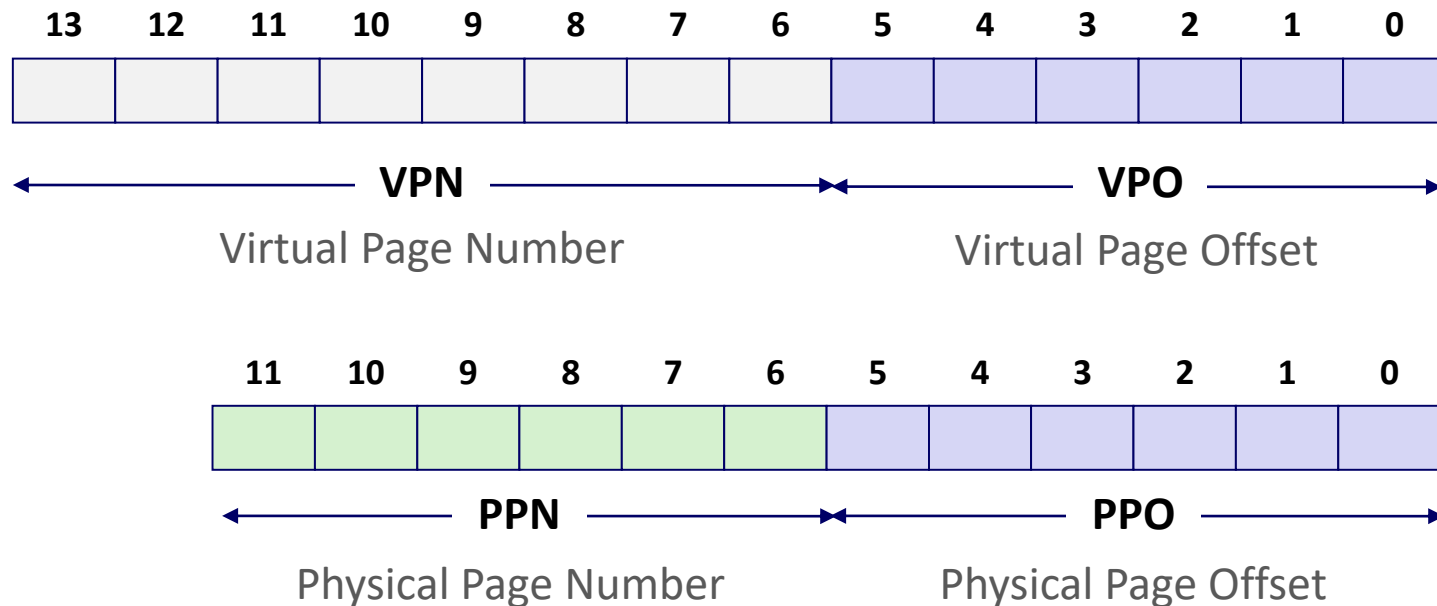
Simple Memory System: Page Table

❖ 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?

	TLB tag						TLB index								
13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**		

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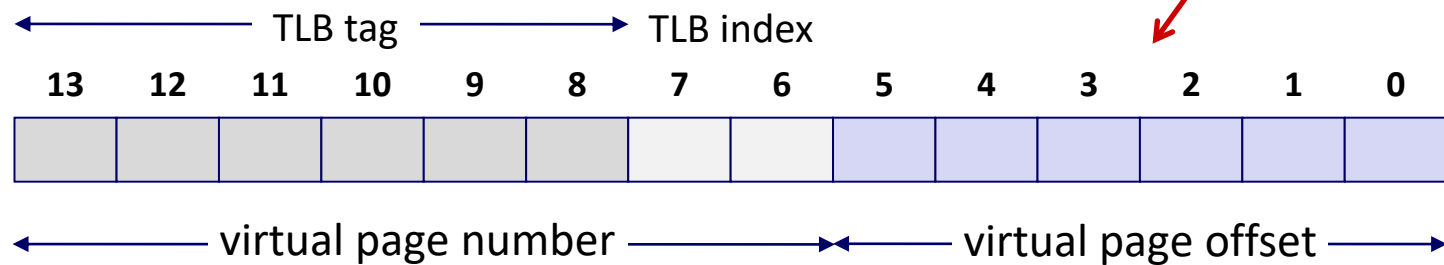
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

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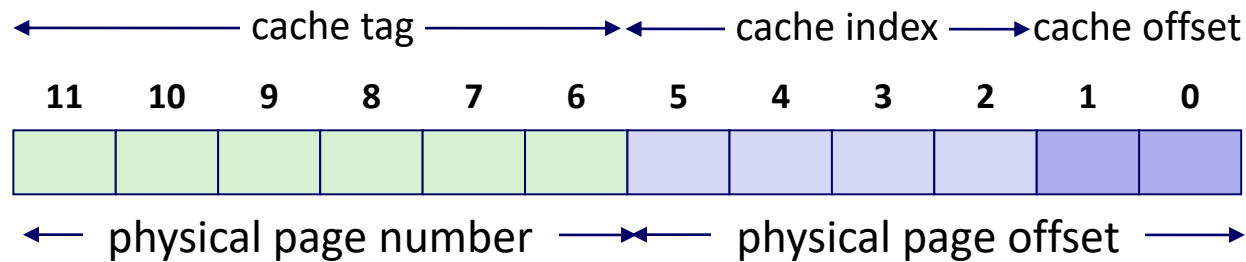
Index	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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

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

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

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`
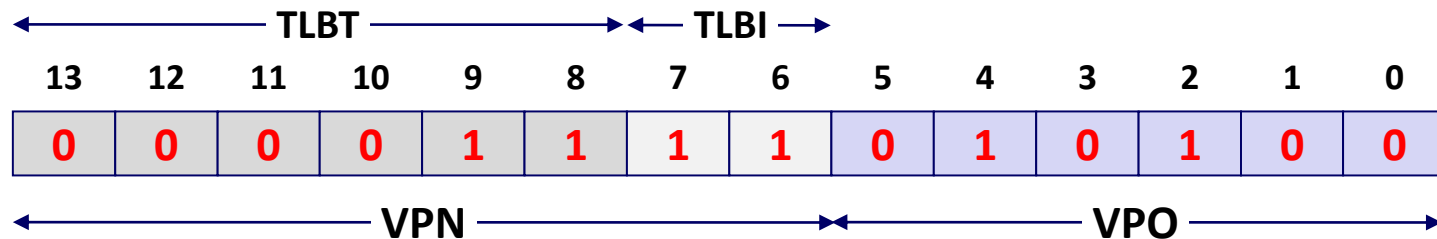
	TLBT →						← TLBI →							
13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	1	1	1	1	0	1	0	1	0	0

← VPN → | ← VPO →

VPN _____   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 →

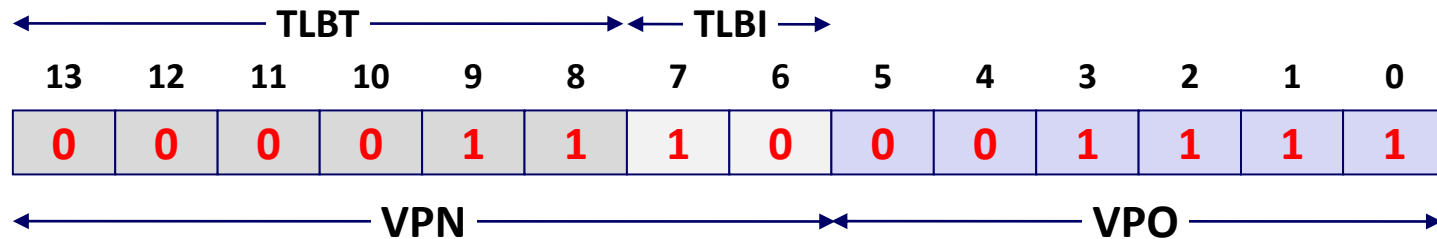CT _____   CI _____   CO _____   Cache Hit? ___   Data (byte) _____

35
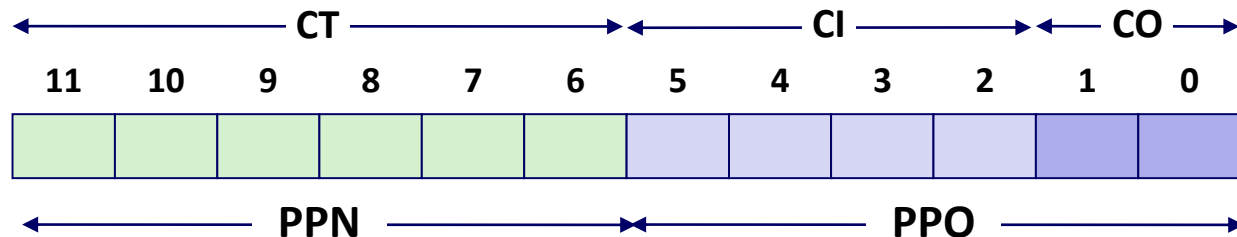
Memory Request Example #2

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:

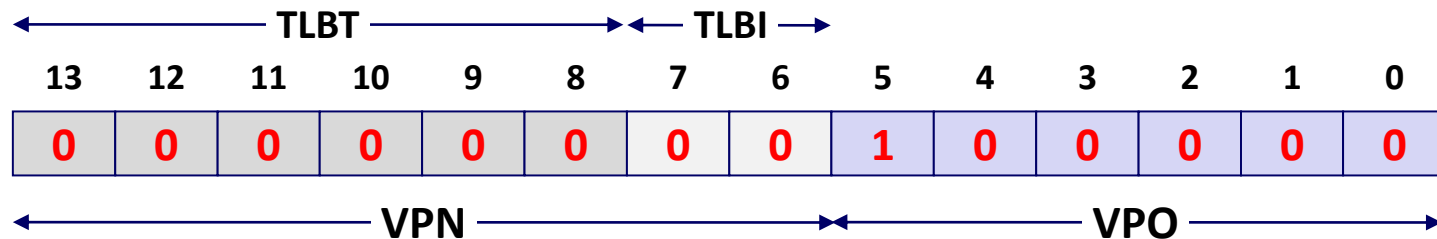|	CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					
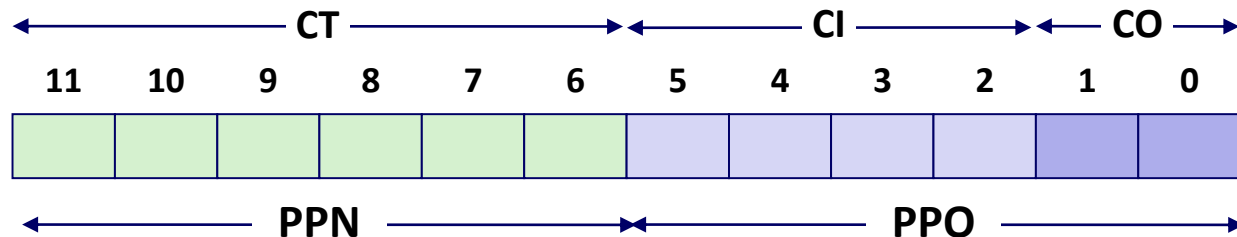
CT _____   CI _____   CO _____   Cache Hit? ___   Data (byte) _____

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 _____   TLBT _____   TLBI _____   TLB Hit? ____   Page Fault? ____   PPN _____

❖ Physical Address:

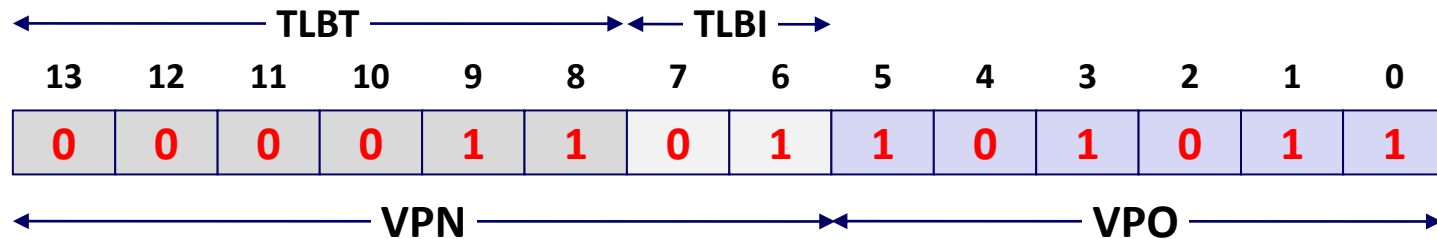|	CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
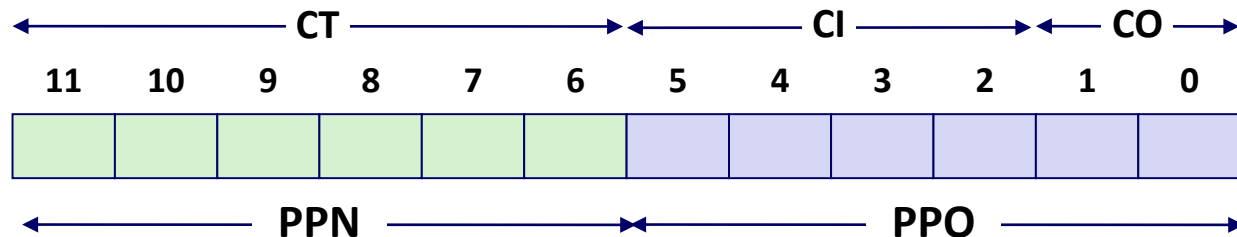
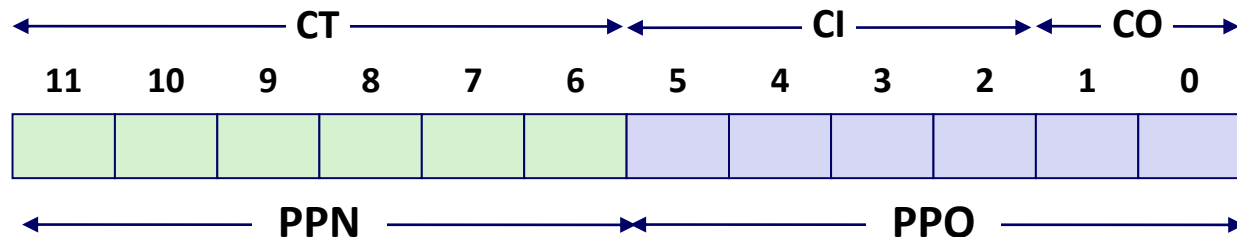CT _____   CI _____   CO _____   Cache Hit? ____   Data (byte) _____

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) _____

Practice VM Question

❖ Our system has the following properties

- 1 MiB of physical address space

- 4 GiB of virtual address space

- 32 KiB page size

- 4-entry fully associative TLB with LRU replacement

a)   Fill in the following blanks:

_____   Entries in a page table   _____   Minimum bit-width of
                                                                         PTBR

_____   TLBT bits   _____   Max # of valid entries
                                                                         in a page table

Practice VM Question

❖ 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)?

Practice VM Question

❖ 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;
```

c) Assuming all of `mat[]` starts on disk, what are the following hit rates for the execution of the for-loop?

_____ TLB Hit Rate   _____ Page Table Hit Rate