

CSE 153

Design of Operating Systems

Winter 2023

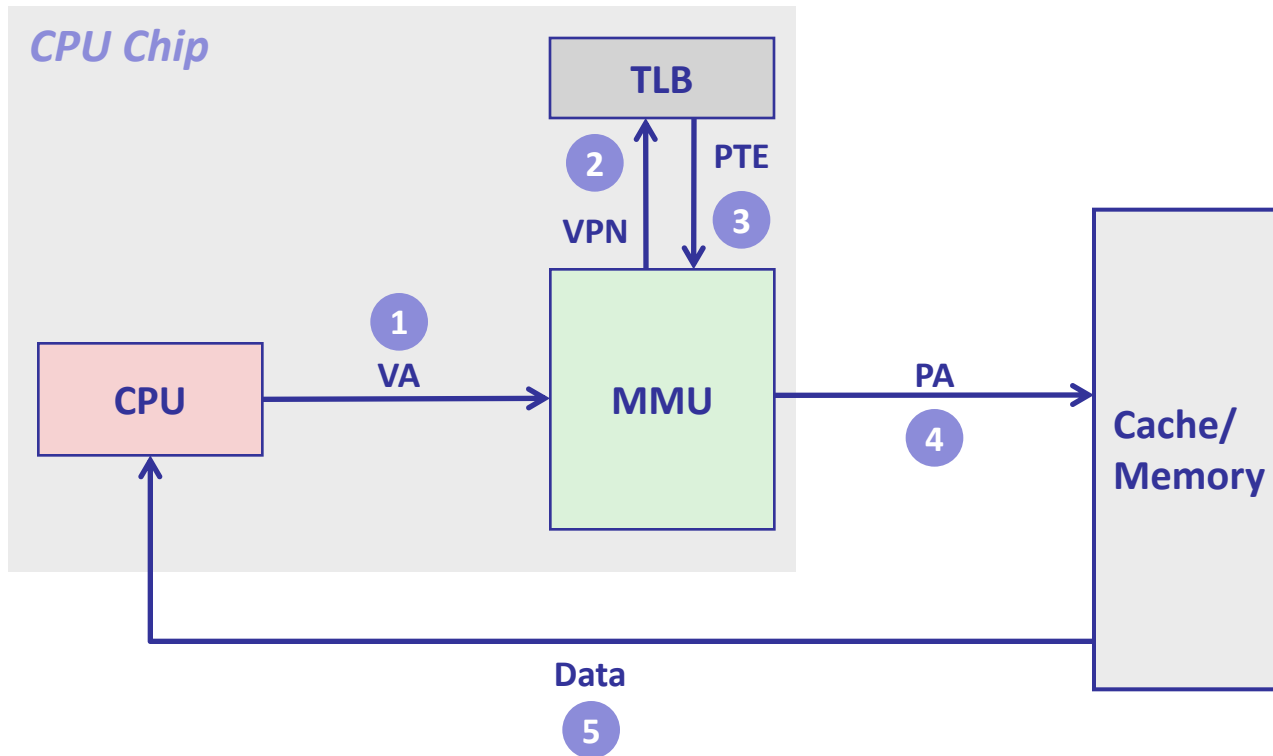
Lecture 17: Virtual Memory (2)
TLBs and Multi-level page tables

Some slides modified from originals by Dave O'hallaron

Speeding up Translation with a TLB

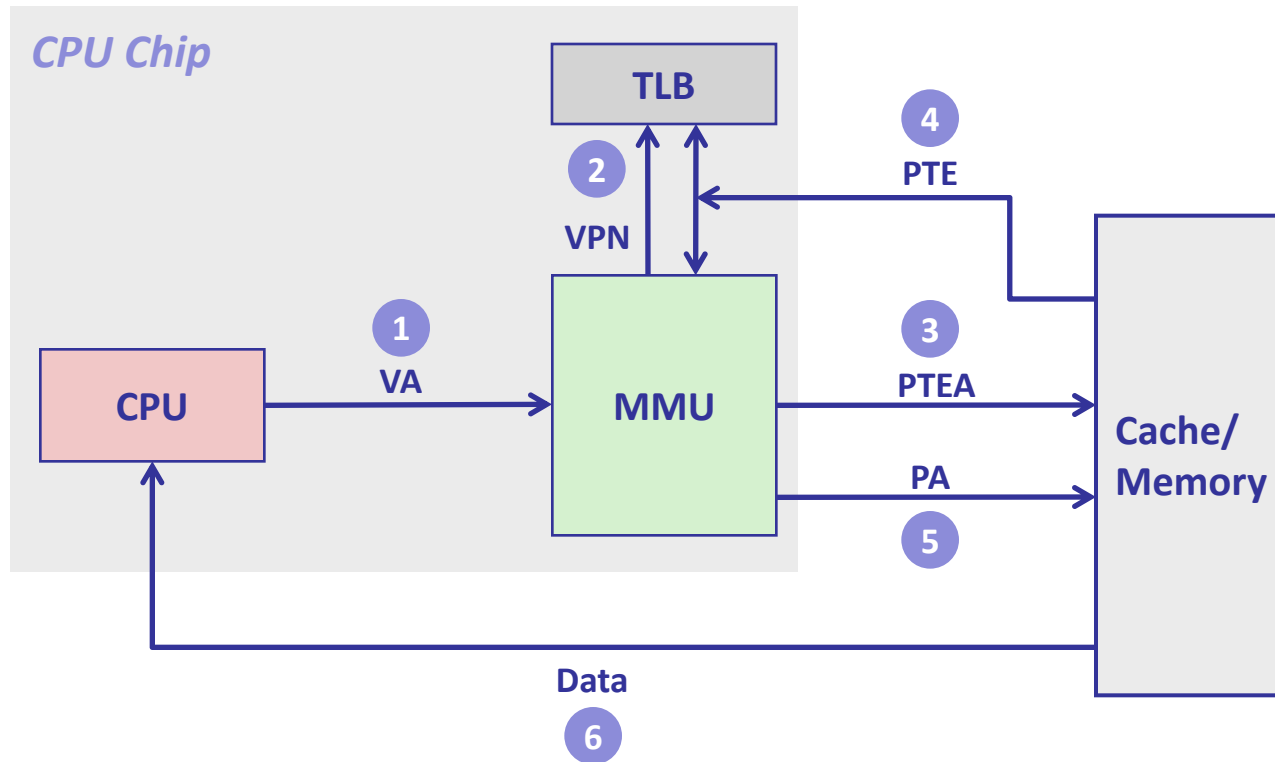
- Page table entries (PTEs) are cached in L1 like any other memory word
 - ◆ PTEs may be evicted by other data references
 - ◆ PTE hit still requires a small L1 delay
- Solution: *Translation Lookaside Buffer* (TLB)
 - ◆ Small hardware cache in MMU
 - ◆ Maps virtual page numbers to physical page numbers
 - ◆ Contains complete page table entries for small number of pages

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. Why?

Reloading the TLB

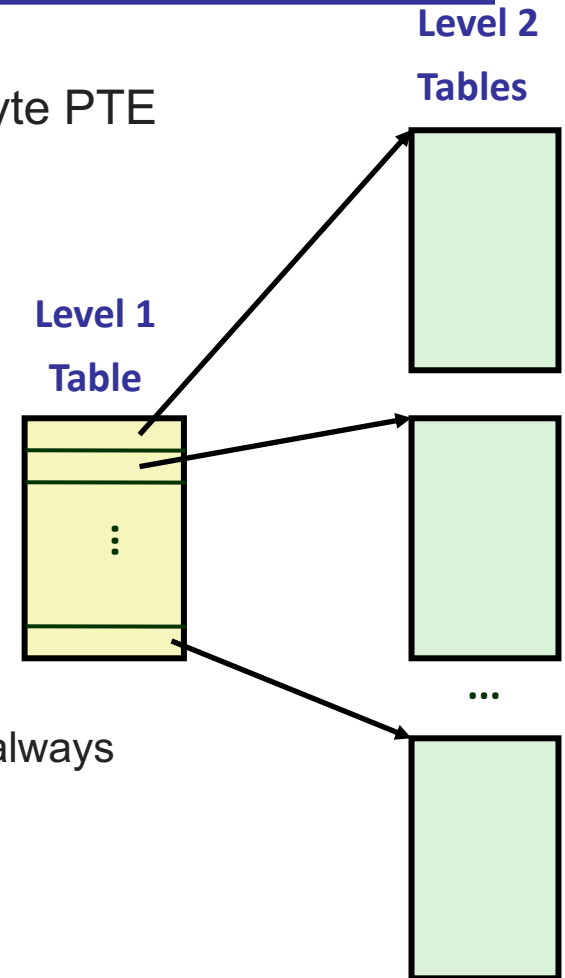
- If the TLB does not have mapping, two possibilities:
 1. MMU loads PTE from page table in memory
 - » Hardware managed TLB, OS not involved in this step
 - » OS has already set up the page tables so that the hardware can access it directly
 2. Trap to the OS
 - » Software managed TLB, OS intervenes at this point
 - » OS does lookup in page table, loads PTE into TLB
 - » OS returns from exception, TLB continues
- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB

Page Faults

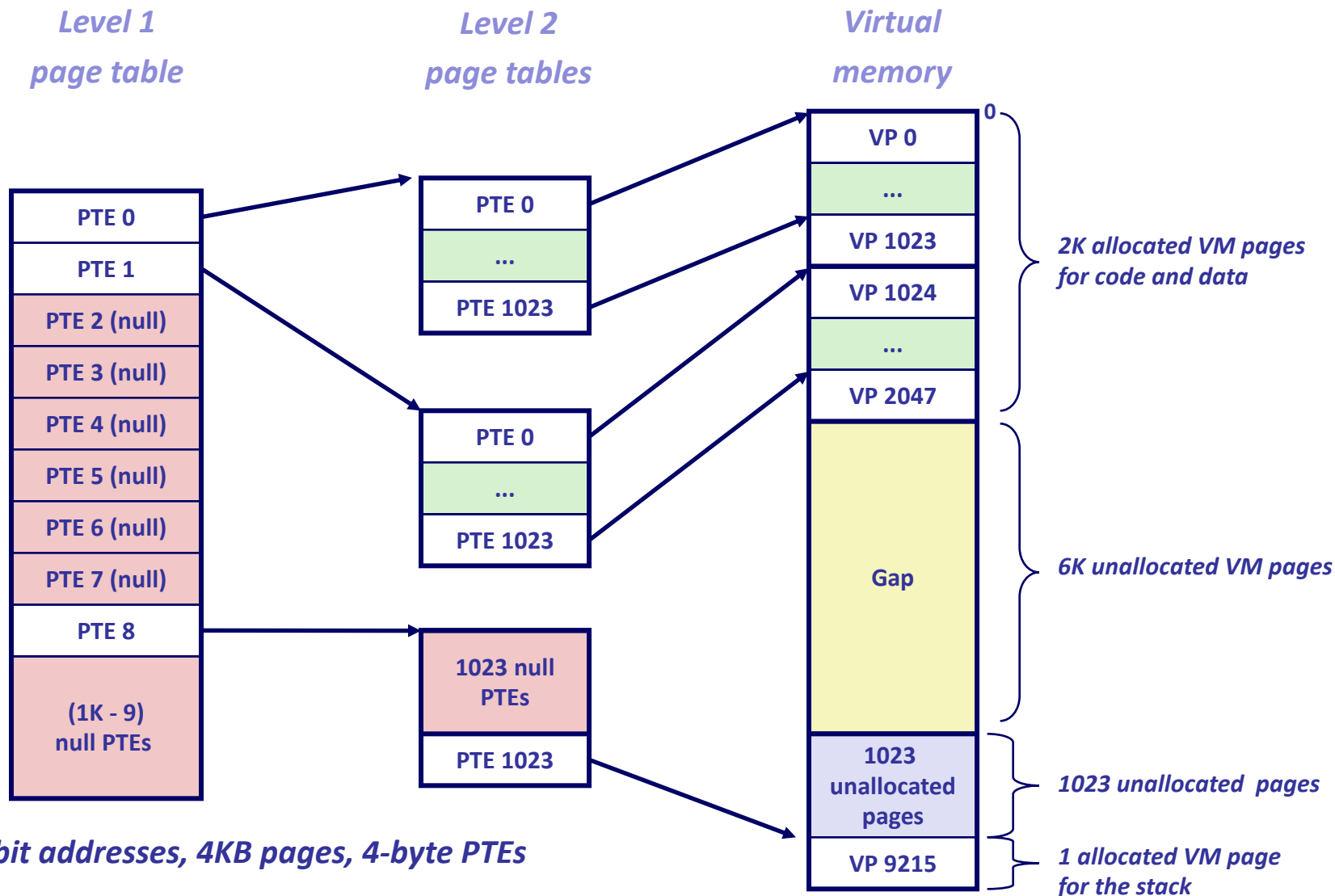
- PTE can indicate a protection fault
 - ◆ Read/write/execute – operation not permitted on page
 - ◆ Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
 - ◆ R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
 - ◆ Invalid
 - » Virtual page not allocated in address space
 - OS sends fault to process (e.g., segmentation fault)
 - » Page not in physical memory
 - OS allocates frame, reads from disk, maps PTE to physical frame

Multi-Level Page Tables

- Suppose:
 - ◆ 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem:
 - ◆ Would need a 512 GB page table!
 - » $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes
- Common solution:
 - ◆ Multi-level page tables
 - ◆ Example: 2-level page table
 - » Level 1 table: each PTE points to a page table (always memory resident)
 - » Level 2 table: each PTE points to a page (paged in and out like any other data)

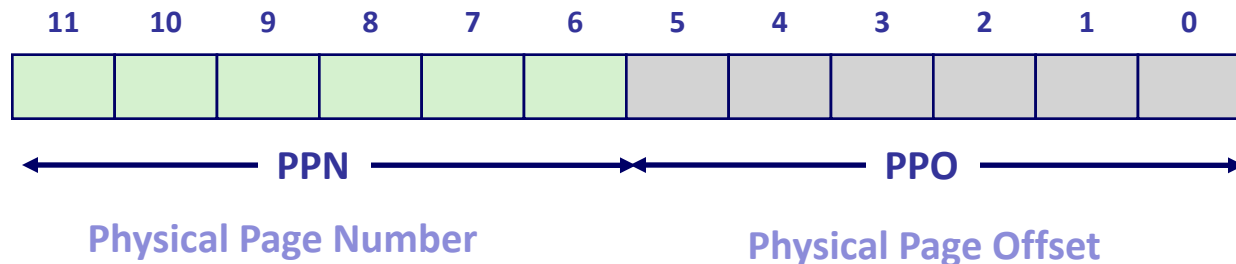
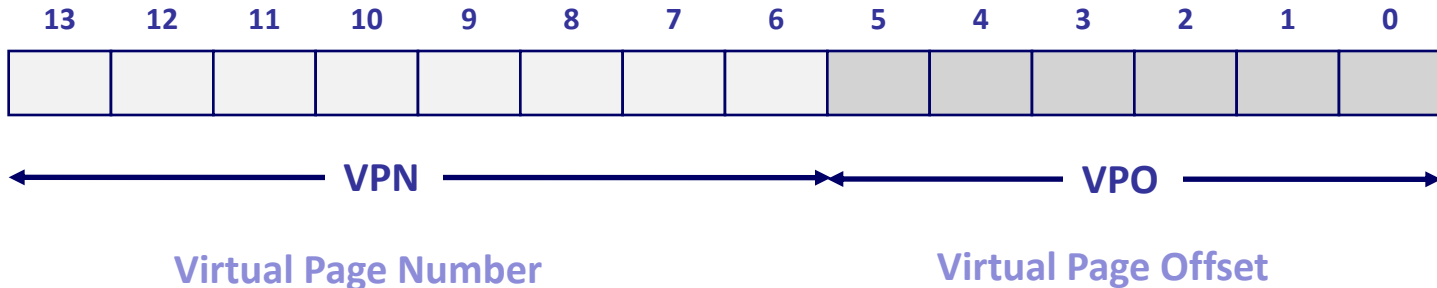


A Two-Level Page Table Hierarchy



Simple Memory System Example

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



Simple Memory System

Page Table

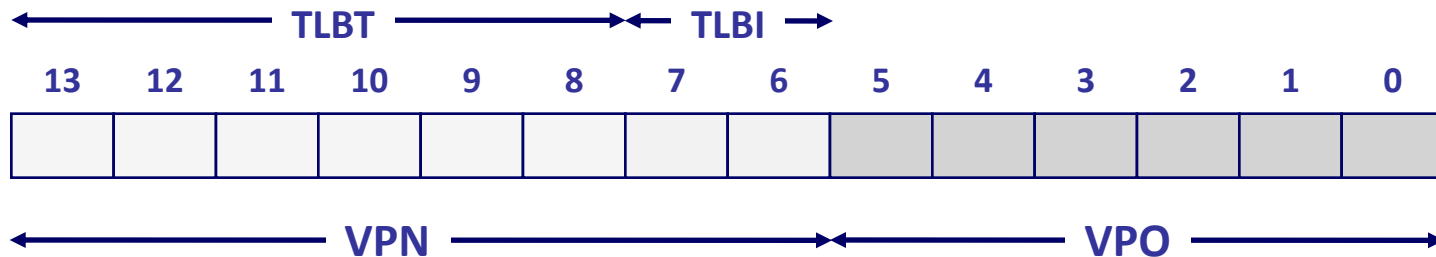
Only show first 16 entries (out of 256)

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

<i>VPN</i>	<i>PPN</i>	<i>Valid</i>
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

Simple Memory System TLB

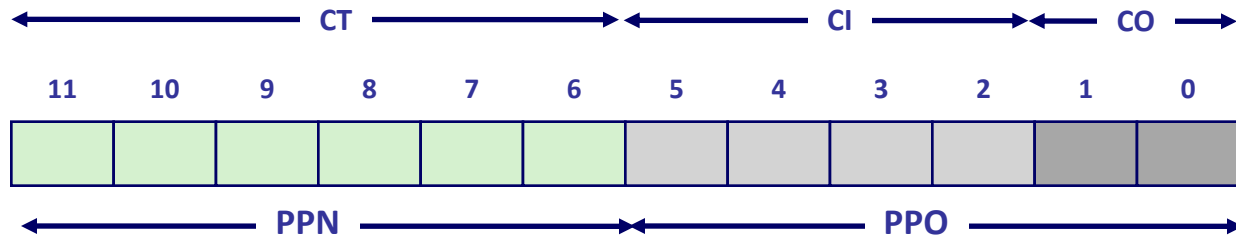
- 16 entries
- 4-way associative



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

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped



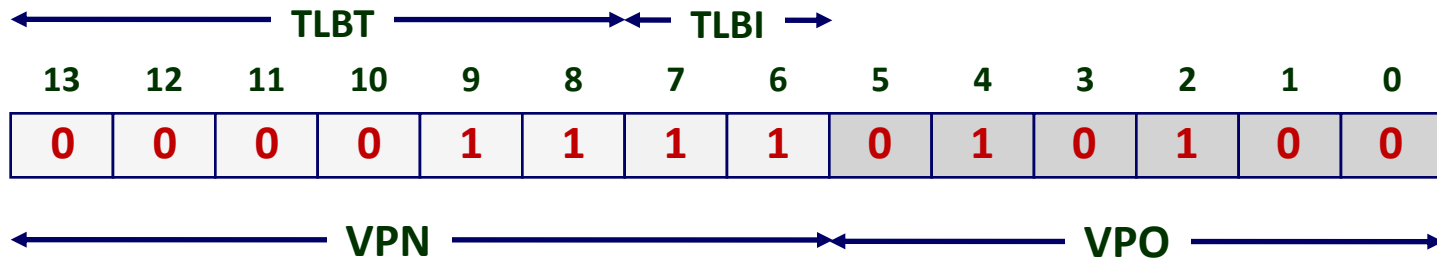
<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

<i>Idx</i>	<i>Tag</i>	<i>Valid</i>	<i>B0</i>	<i>B1</i>	<i>B2</i>	<i>B3</i>
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	-	-	-	-

Address Translation

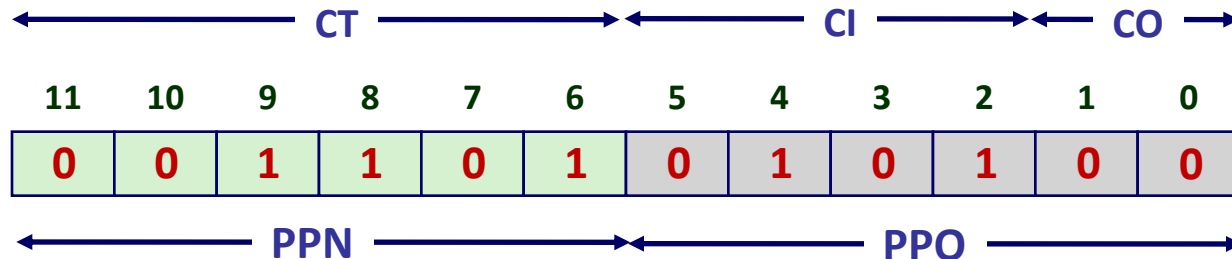
Example #1

Virtual Address: 0x03D4



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

Physical Address

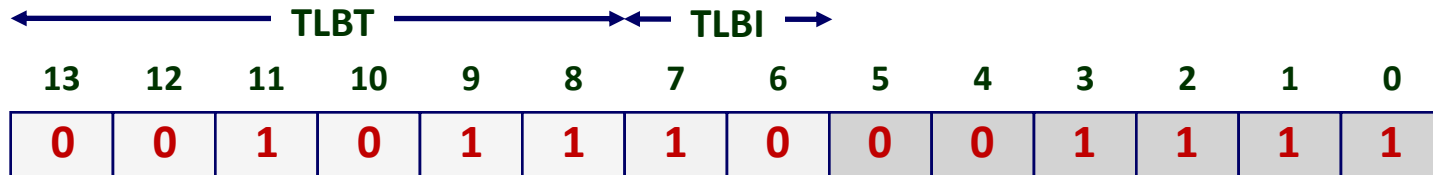


CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

Address Translation

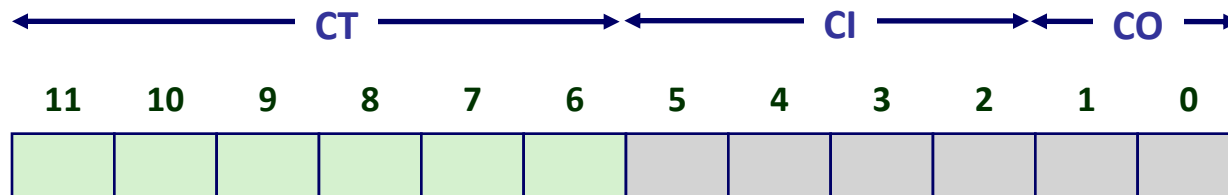
Example #2

Virtual Address: $0 \times 0B8F$



VPN $0 \times 2E$ TLBI 2 TLBT $0 \times 0B$ TLB Hit? N Page Fault? Y PPN: TBD

Physical Address

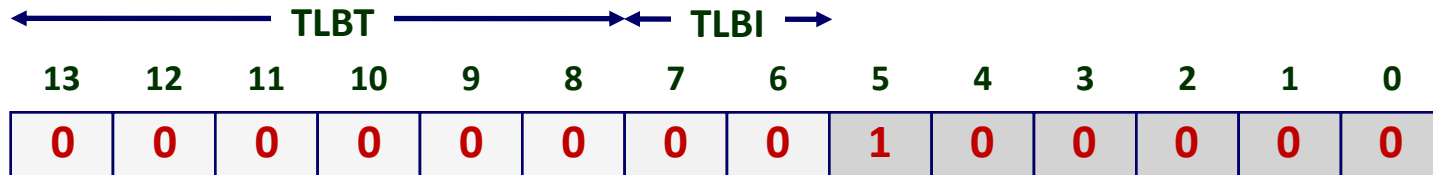


CO ___ CI ___ CT ___ Hit? ___ Byte: ___

Address Translation

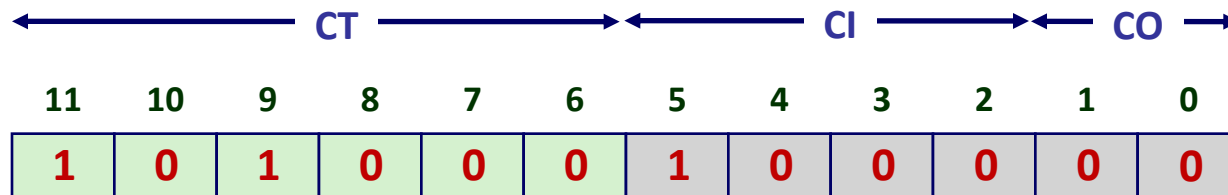
Example #3

Virtual Address: 0x0020



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

Physical Address

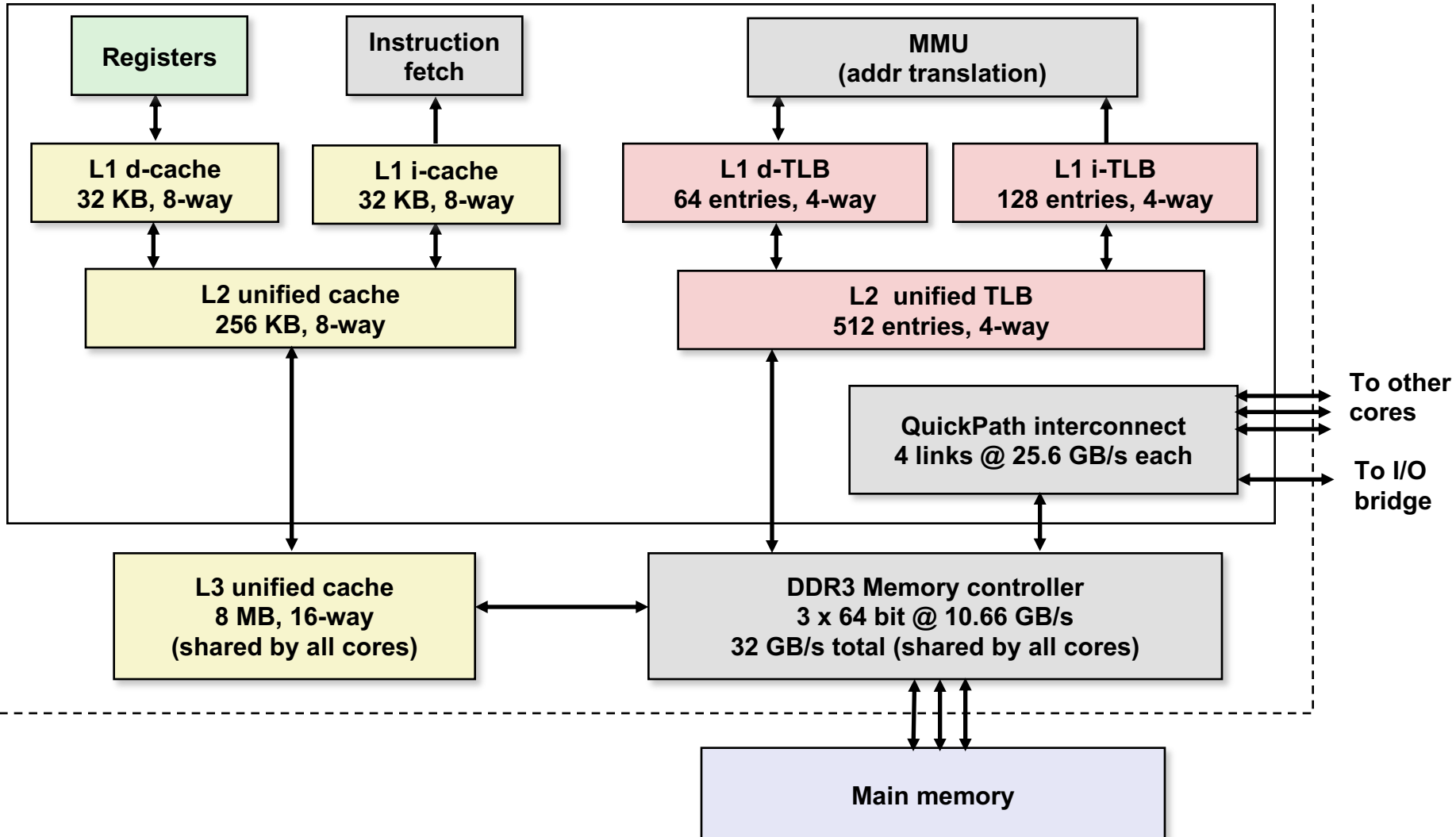


CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

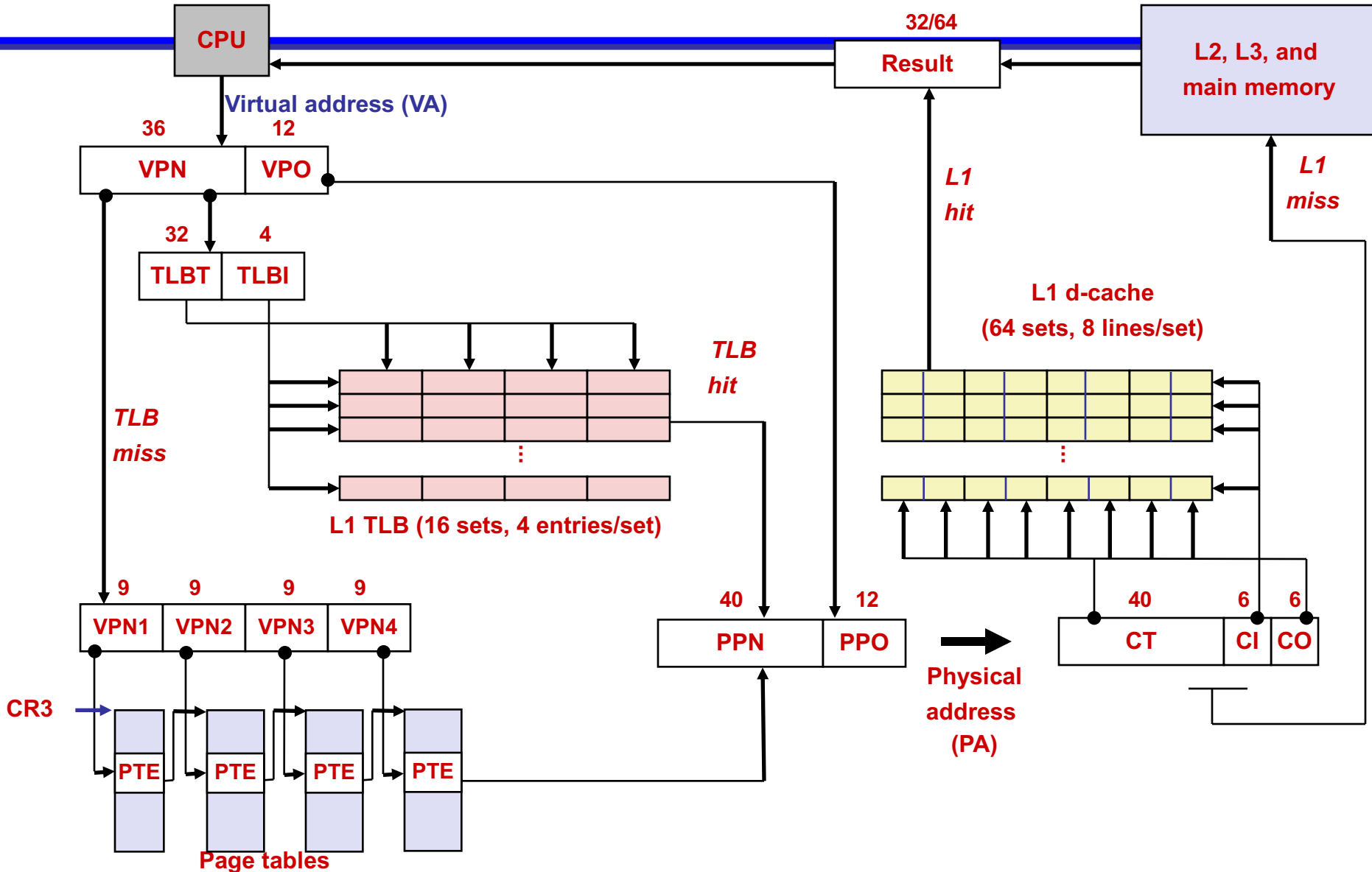
Intel Core i7 Memory System

Processor package

Core x4



End-to-end Core i7 Address Translation



Core i7 Level 1-3 Page Table Entries

63	62	52	51	12	11	9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address				Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)														P=0
--	--	--	--	--	--	--	--	--	--	--	--	--	--	-----

Each entry references a 4K child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don't evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

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 programming
 - ◆ Simplifies protection by providing a convenient interpositioning point to check permissions

Summary (2)

Paging mechanisms:

- Optimizations
 - ◆ Managing page tables (space)
 - ◆ Efficient translations (TLBs) (time)
 - ◆ Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
 - ◆ Sharing memory
 - ◆ Copy on Write
 - ◆ Mapped files

Next time: Paging policies

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
 - ◆ Instead of “open, read into buffer, operate on buffer, ...”
- Bind a file to a virtual memory region (mmap() in Unix)
 - ◆ PTEs map virtual addresses to physical frames holding file data
 - ◆ Virtual address $\text{base} + N$ refers to offset N in file
- Initially, all pages mapped to file are invalid
 - ◆ OS reads a page from file when invalid page is accessed
 - ◆ OS writes a page to file when evicted, or region unmapped
 - ◆ If page is not dirty (has not been written to), no write needed
 - » Another use of the dirty bit in PTE