CPSC313: Computer Hardware and Operating Systems

Unit 5: Virtual Memory

The x86, page faults, and segfaults

Administrivia

- What's happening:
  - Tutorial
  - Lecture today and Friday
  - Quiz 5; last quiz!
  - Normal office hours (through Friday)
  - Exam period office hours (pinned on Piazza) after that
  - Final exam coming soon!
- What's **not** happening
  - Labs are over!
  - No class Wednesday (office hours instead)
  - Fewer in- and pre-class exercises than usual this week (but do the ones we have!)

Today

- Roadmap:
  - Talk through the questions I left you with last time
  - Discuss the different types of faults a VM system exhibits
- Learning Objectives
  - Translate addresses using the x86 page table representation and generalizations of it (e.g., architectures with different sized virtual/physical address spaces and different levels of page tables).
  - Analyze access patterns to determine a number of page faults.
  - Derive the number of bits needed in various (made up) architectures.

Where we left off:

1. Must cr3 contain a physical or virtual address? Why?
2. How large is a single page table? Why?
3. Why do PTEs contain physical page numbers?
4. Why don't PTEs need an offset? (I.e., why is a page number sufficient?)
5. For each level page table, how much memory is reachable?
6. The L1 CI + CO happened to fit in the VPO. Is that necessary? Is it useful in any way?

1. Must cr3 contain a physical or virtual address?

- `cr3` had better contain a physical address.
  - It is needed to translate a virtual address to a physical one
  - If `cr3` contained a virtual address, how would it translate that address?

2. How large is a single page table? Why?

- Each page table has 512 entries
- Each entry is 64 bits (8-bytes)
- Therefore, a page table takes up 4 KB
- Isn’t that convenient, since the page size is also 4 KB!
  - This is no accident!

3. Why do PTEs contain physical page numbers?

- The answer here is pretty similar to that for #1.
- Since we must use the PTEs during address translation, if they contained virtual addresses, we might find ourselves in an infinite loop trying to translate addresses.

4. Why don't PTEs need an offset?

- What is a PTE referencing?
- It is referencing either:
  - A **page** in memory
  - A **page** table in memory – which happens to also be precisely one page large.
- Pages are, by definition, aligned on 4 KB boundaries (because each page is 4 KB).
- Therefore, the 12 low order bits are always 0 for the beginning of a page or a page table.

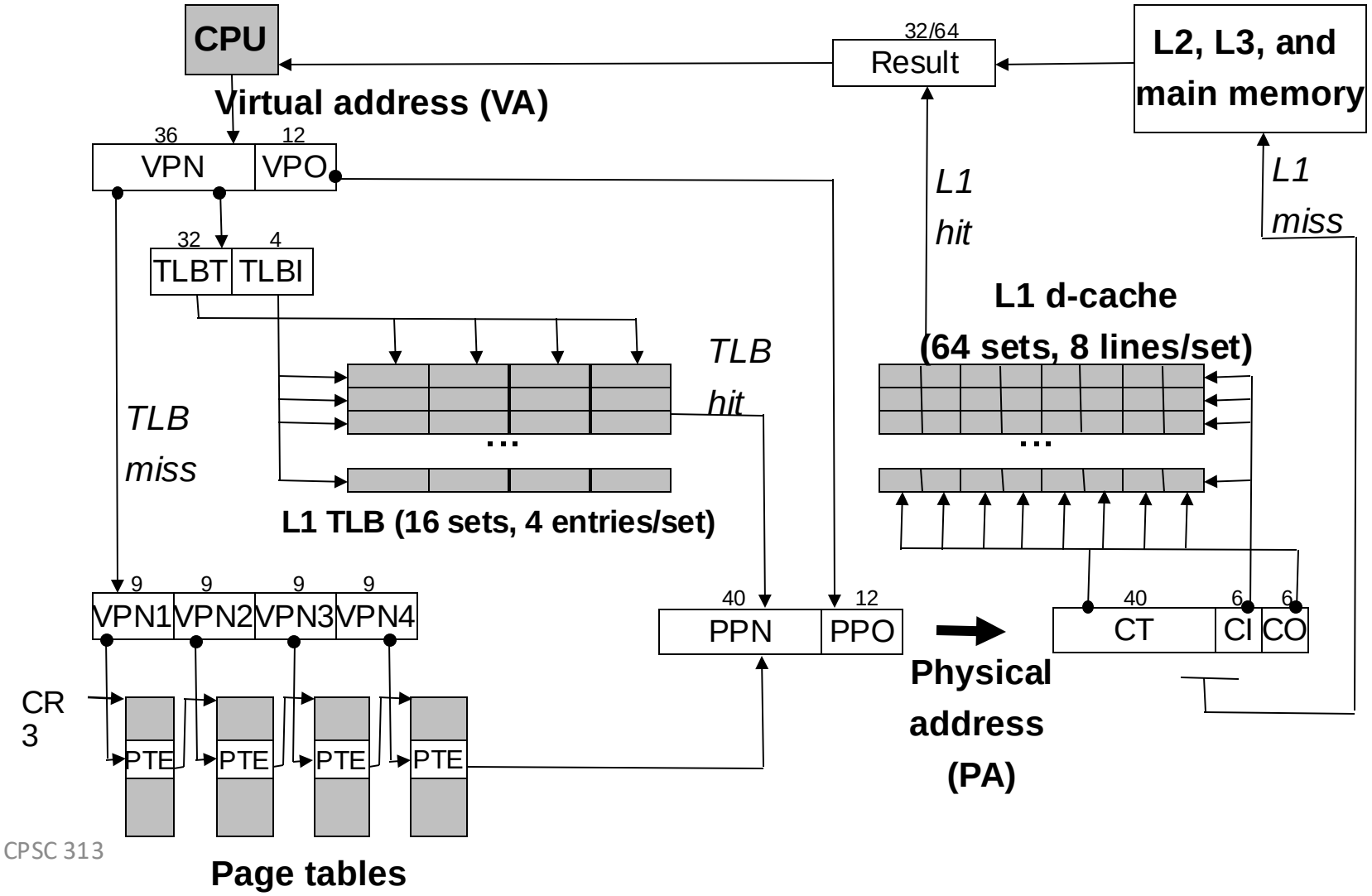
5. For each level page table, how much memory is reachable? (part 1)

- Let's start with L4:
  - Each entry references a single page (of 4 KB)
  - There are 512 entries in a page table
  - A whole L4 page table: 512 * 4 KB = 2 MB.
- L3:
  - Each entry references an entire L4 page table, so an entry reaches 2 MB
  - There are 512 entries.
  - A whole page table: 512 * 2 MB = 1 GB

5. For each level page table, how much memory is reachable? (part 2)

- L2:
  - Each entry references an entire L3 page table, so an entry reaches 1 GB
  - There are 512 entries.
  - A whole page table: 512 * 1 GB = 512 GB (= ½ TB)
- L1:
  - Each entry references an entire L2 page table, so an entry reaches 512 GB
  - There are 512 entries.
  - A whole page table: 512 * 512 GB = 256 TB
- The L1 should access the entire address space. That's 48 bits on the x86-64, and conveniently, 48 bits is 256 TB. Whew!

6. The L1 CI + CO happened to fit in the VPO. Is that necessary? Is it useful in any way?



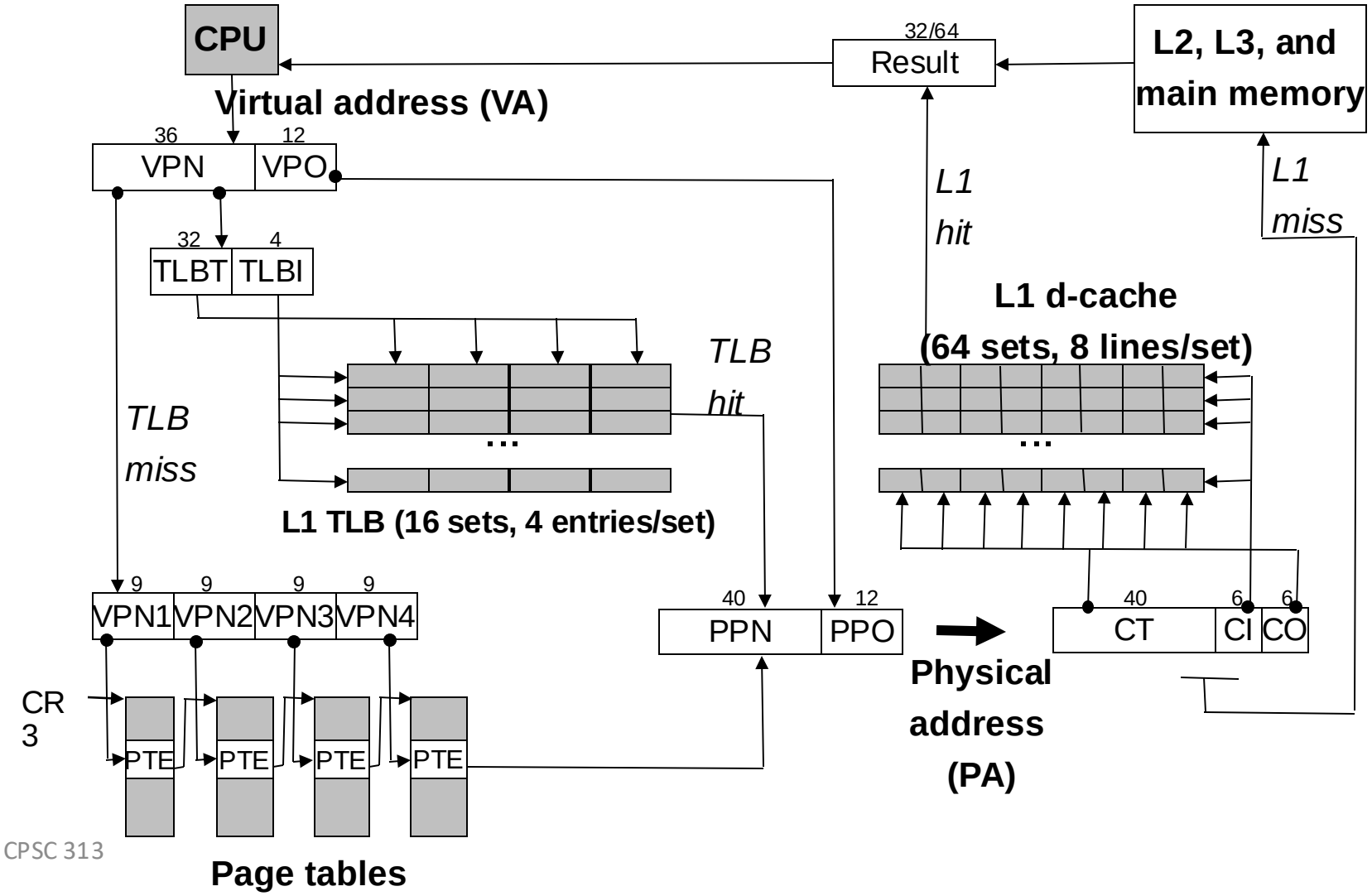
Your L1 cache **might** be able to run its computation faster if you get its index (and offset) to it early.

Can you this computer do so with its 12 bit VPO?

If it's 11 bits?

If it's 13 bits?

6. The L1 CI + CO happened to fit in the VPO. Is that necessary? Is it useful in any way?



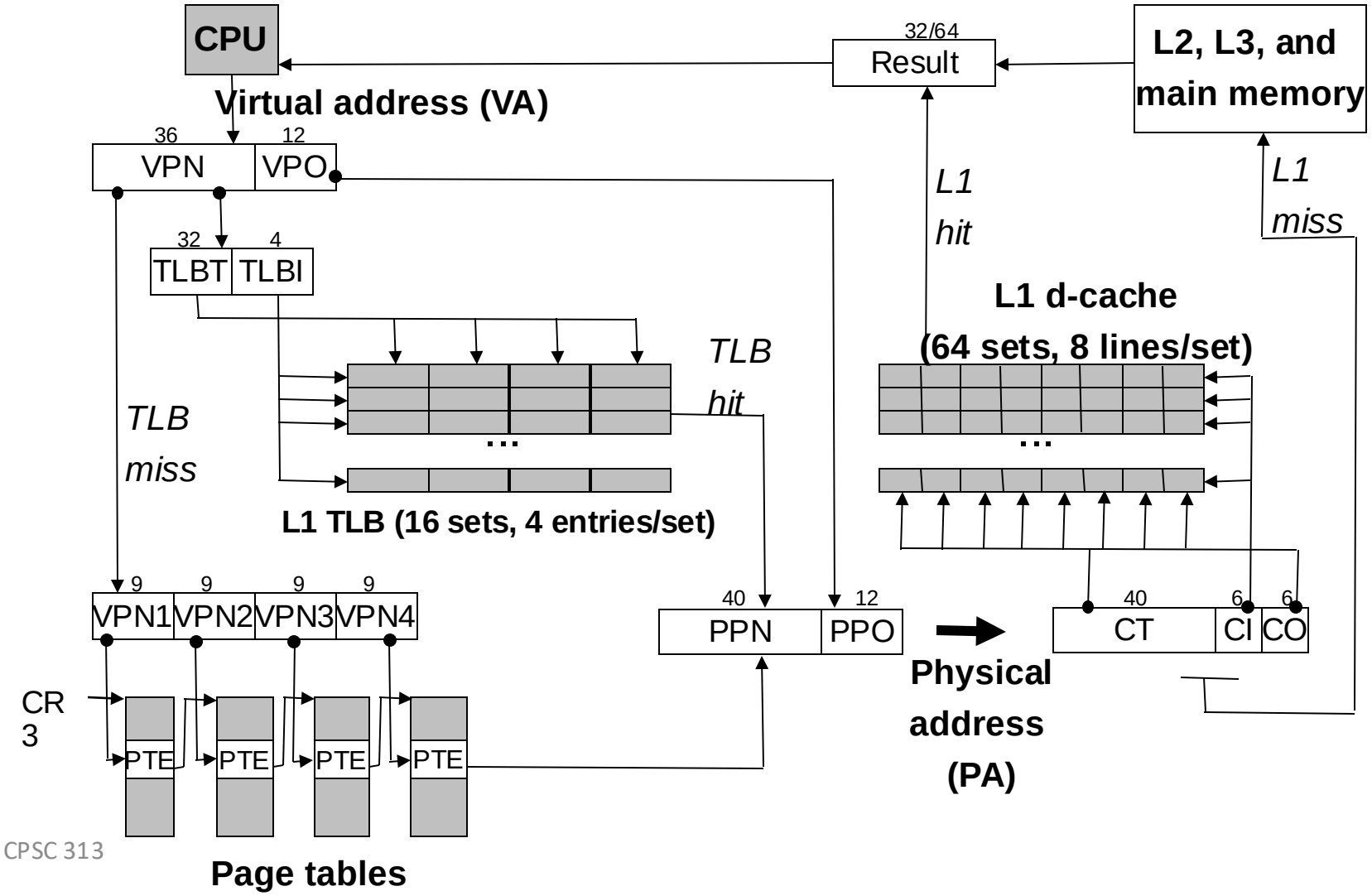
Your L1 cache **might** be able to run its computation faster if you get its index (and offset) to it early.

Can you this computer do so with its 12 bit VPO? Yes, because the VPO and PPO are the same. So we can *immediately* extract CI/CO.

If it's 11 bits?

If it's 13 bits?

6. The L1 CI + CO happened to fit in the VPO. Is that necessary? Is it useful in any way?



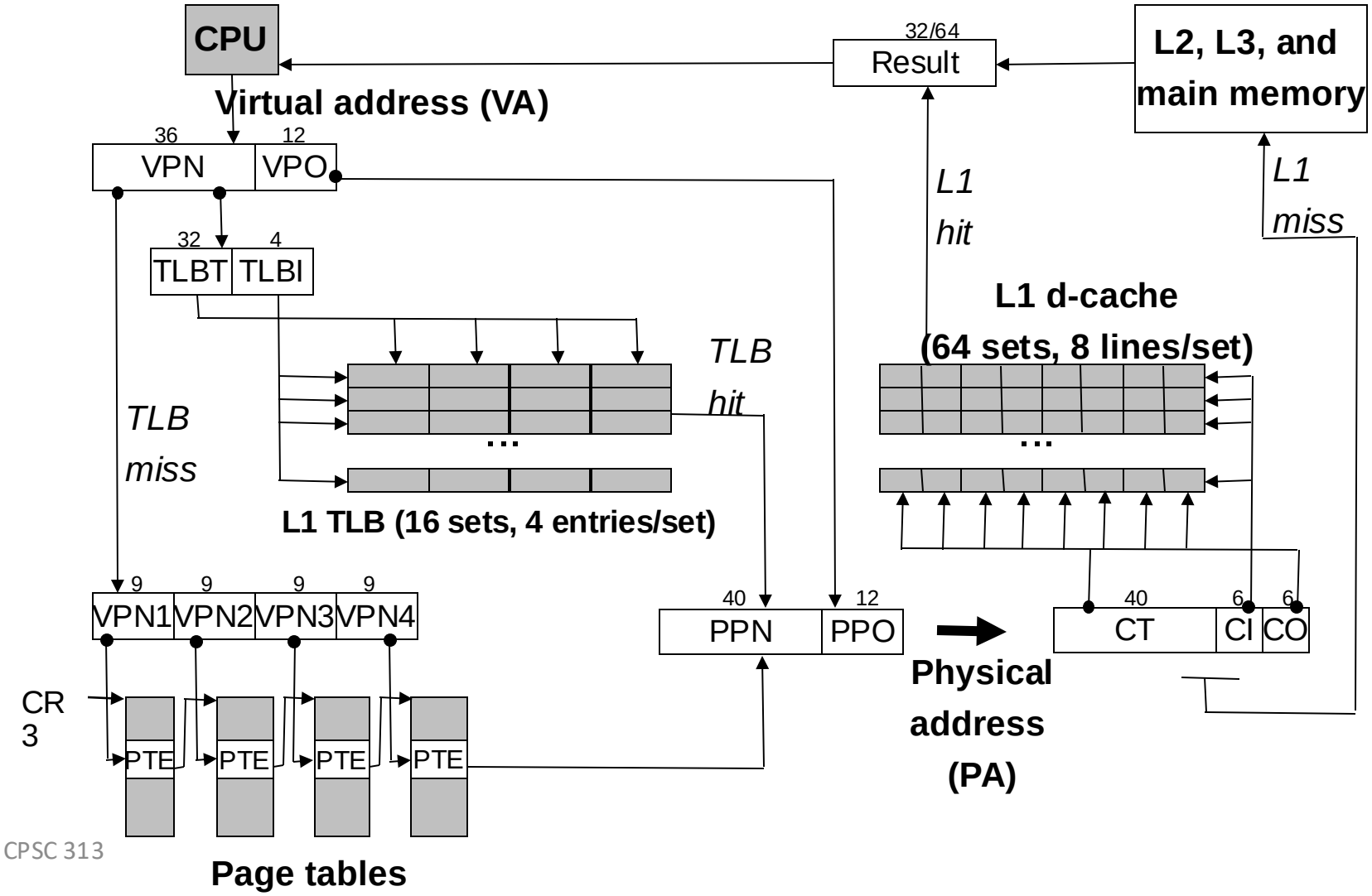
Your L1 cache **might** be able to run its computation faster if you get its index (and offset) to it early.

Can you this computer do so with its 12 bit VPO? Yes, because the VPO and PPO are the same. So we can *immediately* extract CI/CO.

If it's 11 bits? No! 1 bit of the CI is virtual and so unknown until translation finishes.

If it's 13 bits?

6. The L1 CI + CO happened to fit in the VPO. Is that necessary? Is it useful in any way?



Your L1 cache **might** be able to run its computation faster if you get its index (and offset) to it early.

Can you this computer do so with its 12 bit VPO? Yes, because the VPO and PPO are the same. So we can *immediately* extract CI/CO.

If it's 11 bits? No! 1 bit of the CI is virtual and so unknown until translation finishes.

If it's 13 bits? Sure. You could even do a tiny bit of tag comparison as well.

x86 Page Table Wrap Up

- Page tables are the data structure that maps from virtual addresses to physical addresses.
- The x86-64 implements 4-level page tables to conserve memory.
- While a complete flat page table would require 512 GB of memory, a tiny process can make do with far less using 4-level page tables:
  - 1 L1 page table (4 KB)
  - 1 L2 page table (4 KB)
  - 1 L3 page table (4 KB)
  - 1 L4 page table (4 KB)
  - Total: 16 KB

Virtual Memory System Faults (1)

- Page Fault:
  - The virtual page being accessed is a **valid page (i.e., previously allocated virtual page)**, but it is not in memory.
    - Does the process get to keep running?

Virtual Memory System Faults (1)

- Page Fault:
  - The virtual page being accessed is a **valid page (i.e., previously allocated virtual page)**, but it is not in memory.
    - Does the process get to keep running? Yes!
    - How?

Virtual Memory System Faults (1)

- Page Fault:
  - The virtual page being accessed is a **valid page (i.e., previously allocated virtual page)**, but it is not in memory.
    - Does the process get to keep running? Yes!
    - How?
      - The Operating System interprets the PTE
      - It brings the page into memory (likely evicting something.. more soon!)
      - It updates the PTE

Virtual Memory System Faults (2)

- Page Fault:
  - The process tries to write to a page whose write bit is 0 (i.e. that it is not allowed to write to)
    - Does the process get to keep running?

Virtual Memory System Faults (2)

- Page Fault:
  - The process tries to write to a page whose write bit is 0 (i.e. that it is not allowed to write to)
    - Does the process get to keep running? Maybe...
    - It depends on the situation.
      - This is a fault; so the OS gets to “jump in” and determine the situation. Based on extra data structures the OS maintains.

Virtual Memory System Faults (2)

- Page Fault: Writing to a page whose write bit is 0.
  - Case 1: The process is only allowed to read the page (for instance, a shared library).
    - The operating system terminates the process.

Virtual Memory System Faults (2)

- Page Fault: Writing to a page whose write bit is 0.
  - Case 2: The page is shared with another process with Copy-on-Write (CoW)
    - This happens when the fork() system call creates a child process.
    - The OS does not create a new copy of the data right away.
    - It waits until a page is written to before creating a copy of it.
    - Once it has created the copy, both the parent and the child's write bits are set to 1.

Virtual Memory System Faults (3)

- Segmentation Violation/Segfault:
  - The virtual page being accessed is invalid (i.e., there is nothing allocated to that part of the address space).

Virtual Memory System Faults (3)

- Segmentation Violation/Segfault:
  - The virtual page being accessed is invalid (i.e., there is nothing allocated to that part of the address space).
    - Does the process get to keep running?

Virtual Memory System Faults (3)

- Segmentation Violation/Segfault:
  - The virtual page being accessed is invalid (i.e., there is nothing allocated to that part of the address space).
    - Does the process get to keep running? No!
    - Why can't the OS let the process recover?
      - The process has requested a non-existent object.
      - The OS has no way to signal that the 'load instruction' should return an error.
      - *Or can we just make something up? Well, a fun read:* https://people.csail.mit.edu/rinard/paper/osdi04.pdf

Virtual Memory System Faults (3)

- Segmentation Violation/Segfault:
  - The virtual page being accessed is invalid (i.e., there is nothing allocated to that part of the address space).
    - What does the operating system do?
      - Kills the process.
  - Test your knowledge: Why do you get “segfault core dump”: when you dereference a NULL pointer?

Counting page faults

- Remember questions like this from caching?

  Let's say that you have a 4 KB cache with 64-byte cache lines. Given an 8192-byte array, if you access every 4^{th} byte in the array just once, what will your hit rate be (expressed as a percent, rounded to the nearest integer)?

Counting page faults

- Remember questions like this from caching?

  Let's say that you have a 4 KB cache with 64-byte cache lines. Given an 8192-byte array, if you access every 4th byte in the array just once, what will your hit rate be (expressed as a percent, rounded to the nearest integer)?

- Let's turn it into a VM question:

  Assume a 1024-byte virtual page size. Let's say that you have ~~a 4 KB cache with 64-byte cache lines. Given~~ an 8192-byte array and none of its data is in memory, if you access every 4th byte in the array just once, ~~what will your hit rate be (expressed as a percent, rounded to the nearest integer)~~ how many page faults will you experience?

Calculating page faults

- Remember questions like this from caching?

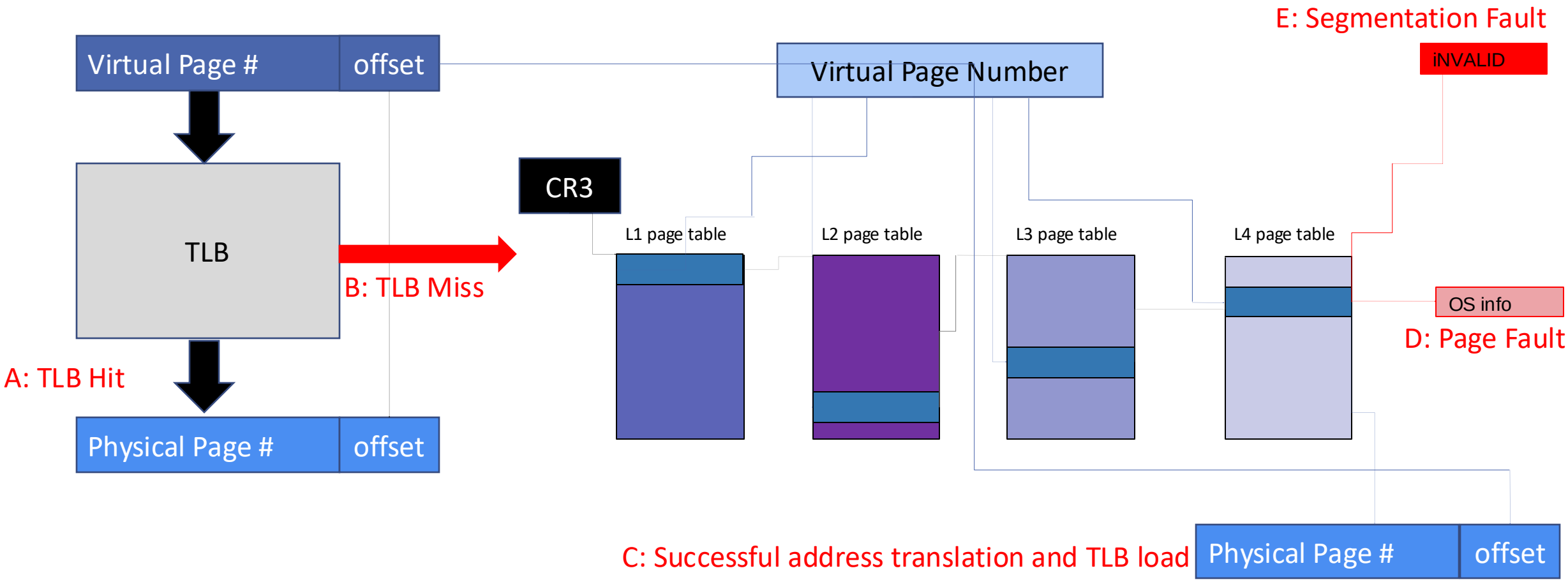
  Let's say that you have a 4 KB cache with 64-byte cache lines. Given an 8192-byte array, if you access every 4^{th} byte in the array just once, what will your hit rate be (expressed as a percent, rounded to the nearest integer)?

- Let's turn it into a VM question:

  Assume a 1024-byte virtual page size. Let's say that you have an 8192-byte array and none of its data is in memory, if you access every 4^{th} byte in the array just once, how many page faults will you experience?

    If pages are 1024 bytes, and your array is 8192 bytes, then you will access 8 pages and you will take a page fault for each one.

Address Translation in One Slide



Reminder

- Next (and final) normal class session:
  - We will talk about page replacement and clock.
  - We will all say farewell until the final!