24. Virtual Memory: TLB and Caches

EECS 370 – Introduction to Computer Organization – Fall 2020

Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Final Exam

Online exam through Gradescope

- Practice exam on Gradescope will be made available

Topics

- Strong emphasis on topics since the midterm
  - Pipelining
  - Branch prediction
  - Caches
  - Virtual memory

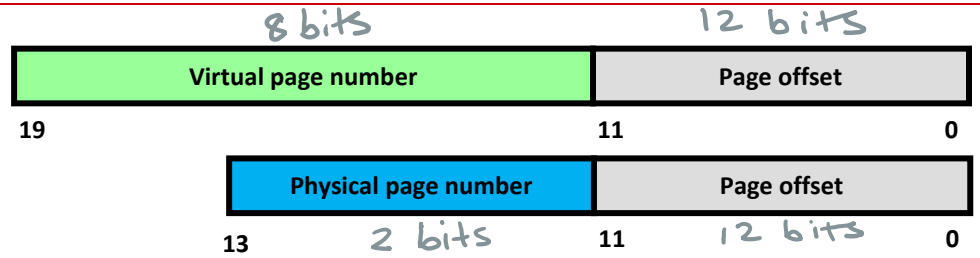
Page offset size = log(4 KB) = 12 bits

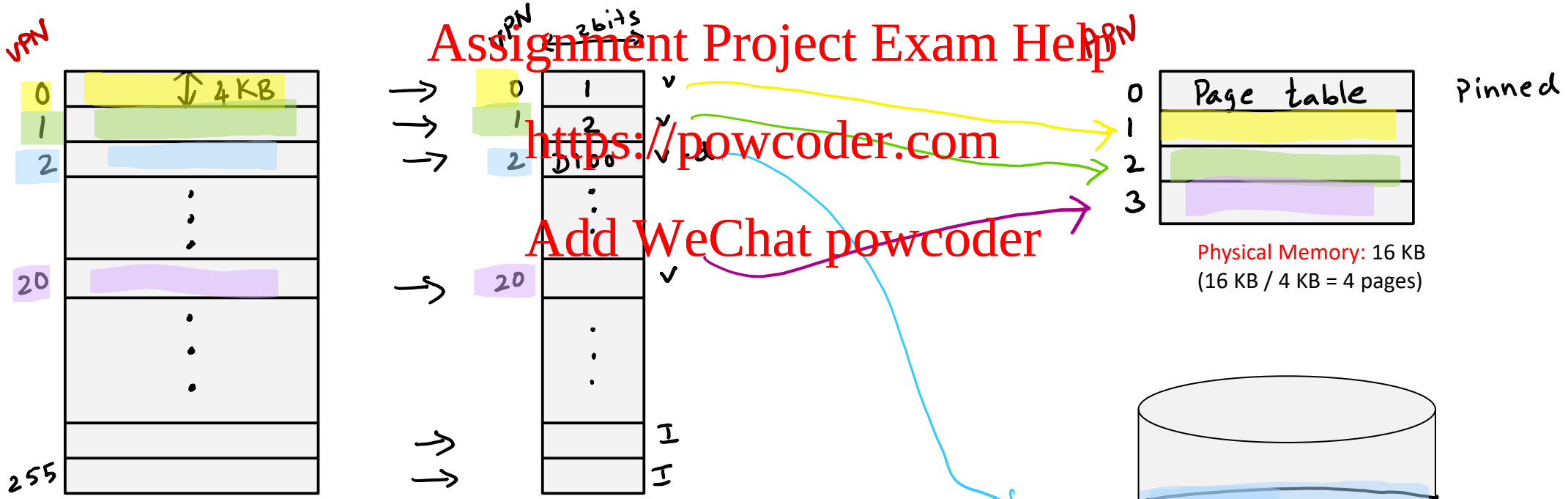
Virtual Memory: An Example

Page size = 4 KB

Virtual memory
(2^20 bytes = 256 pages)

Physical Memory
(16 KB = 4 pages)





Virtual memory: 2^20 bytes

2^20 / 4 KB = 256 pages

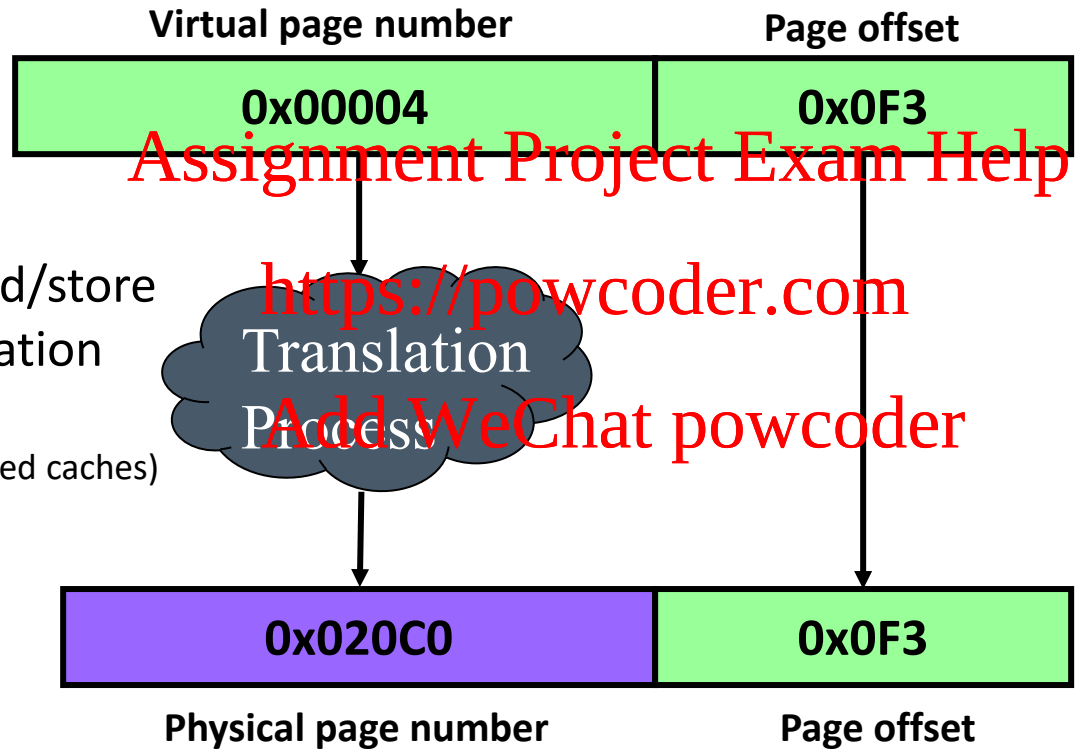
Page Table
(256 entries)

Physical Memory: 16 KB
(16 KB / 4 KB = 4 pages)

Disk
(swap partition)

Address Translation

Virtual address = 0x000040F3



Every instruction fetch, load/store needs to do address translation

(optimized later with virtually-addressed caches)

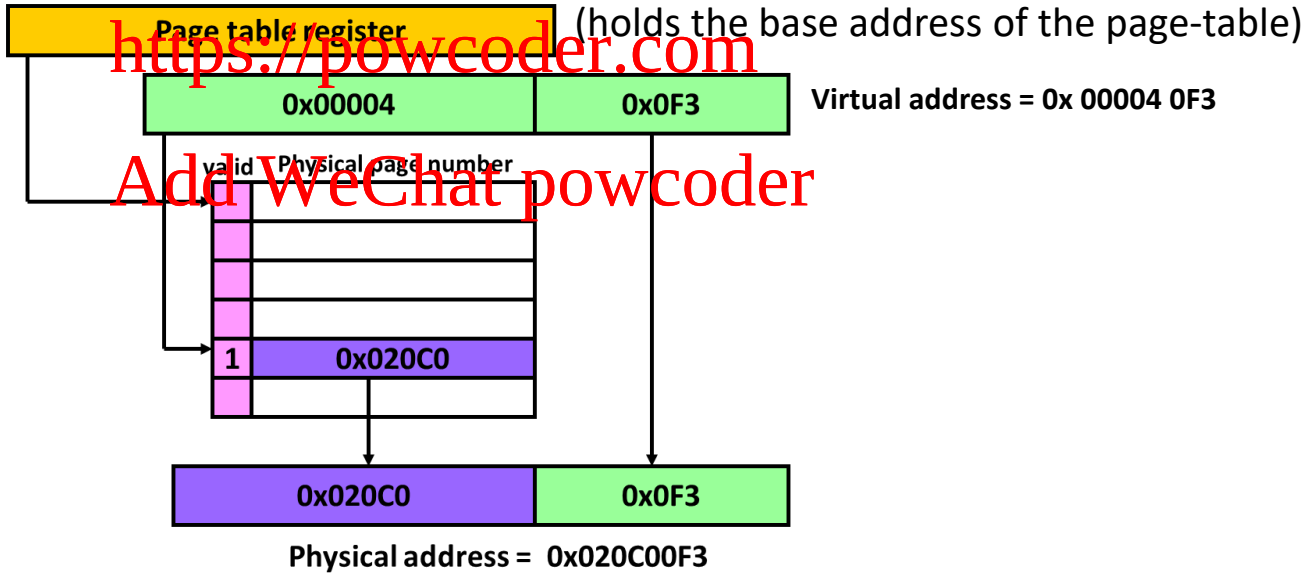
Physical address = 0x020C00F3

Page table lookups for address translation

N-level page table

Each address translation requires N memory accesses, one per level

Single-level page table



Problem: Address translation overhead

Address translation is on the critical path

- Can be done only after virtual address is known
- Need to done before accessing memory (optimized later with virtually-addressed caches)

Address translation requires accesses to the page table(s) in physical memory

- A memory access (instruction fetch, load/store) performs **N additional memory accesses**, where N is the number of levels in a hierarchical page table

- **Slow**

After address translation, memory hierarchy is accessed to perform memory access

Solution: Translation look-aside buffer (TLB)

TLB is a special cache for page-tables.

Speeds-up address translation by reducing main memory accesses to page tables.

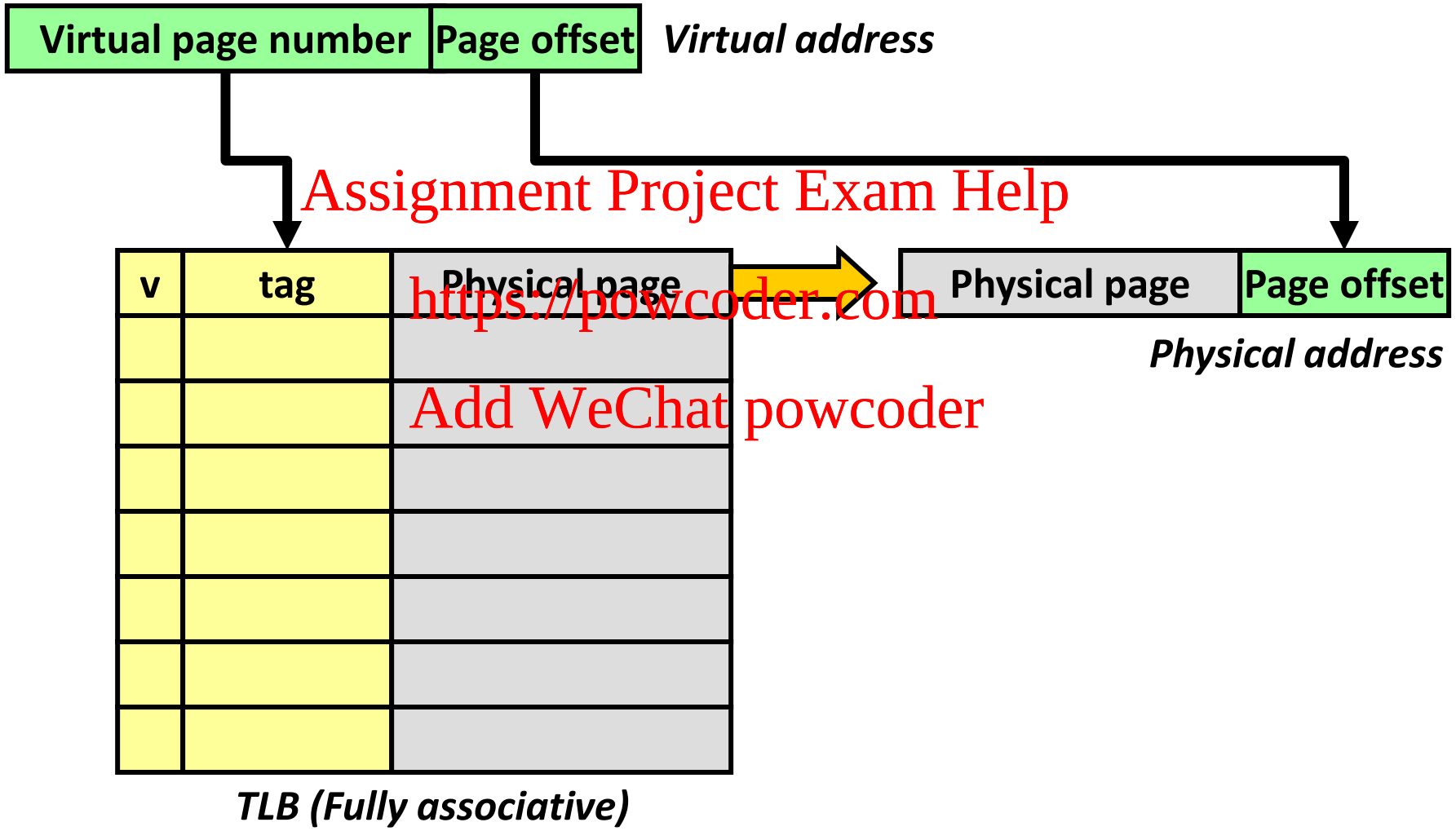
On a TLB miss, access page-table(s) in main memory

Stores a small subset of valid page table entries.

16-512 entries common.

Typically, has low miss rate (< 1%).

Translation look-aside buffer (TLB)



Putting it all together

OS: loading program in memory

Creates a new process P

Constructs a page table for P

Marks all page table entries as invalid with a pointer to the disk image of the program

That is, point to the executable file containing the binary.

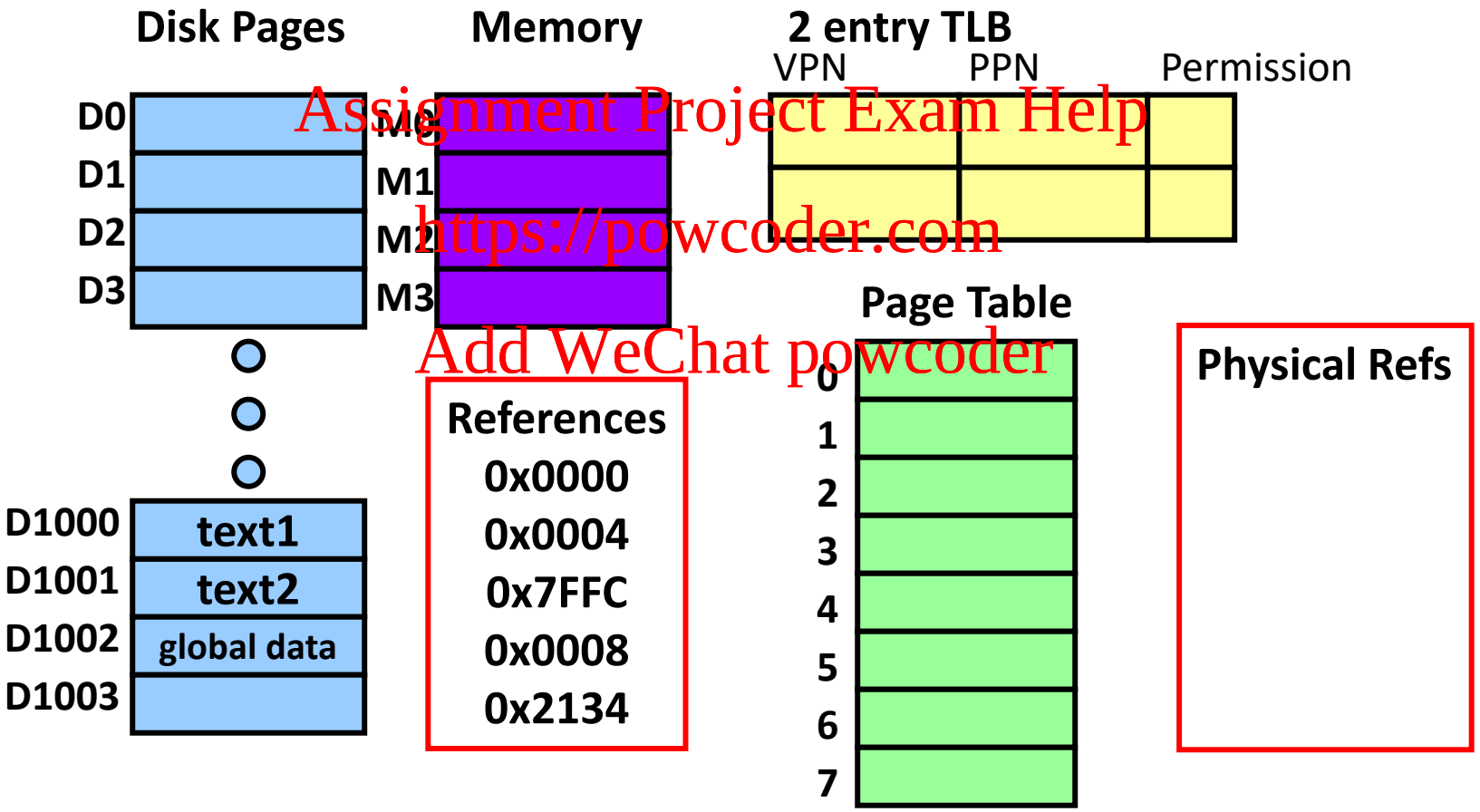
Runs the program

Will get an immediate page fault on the first instruction (everything is on disk initially)

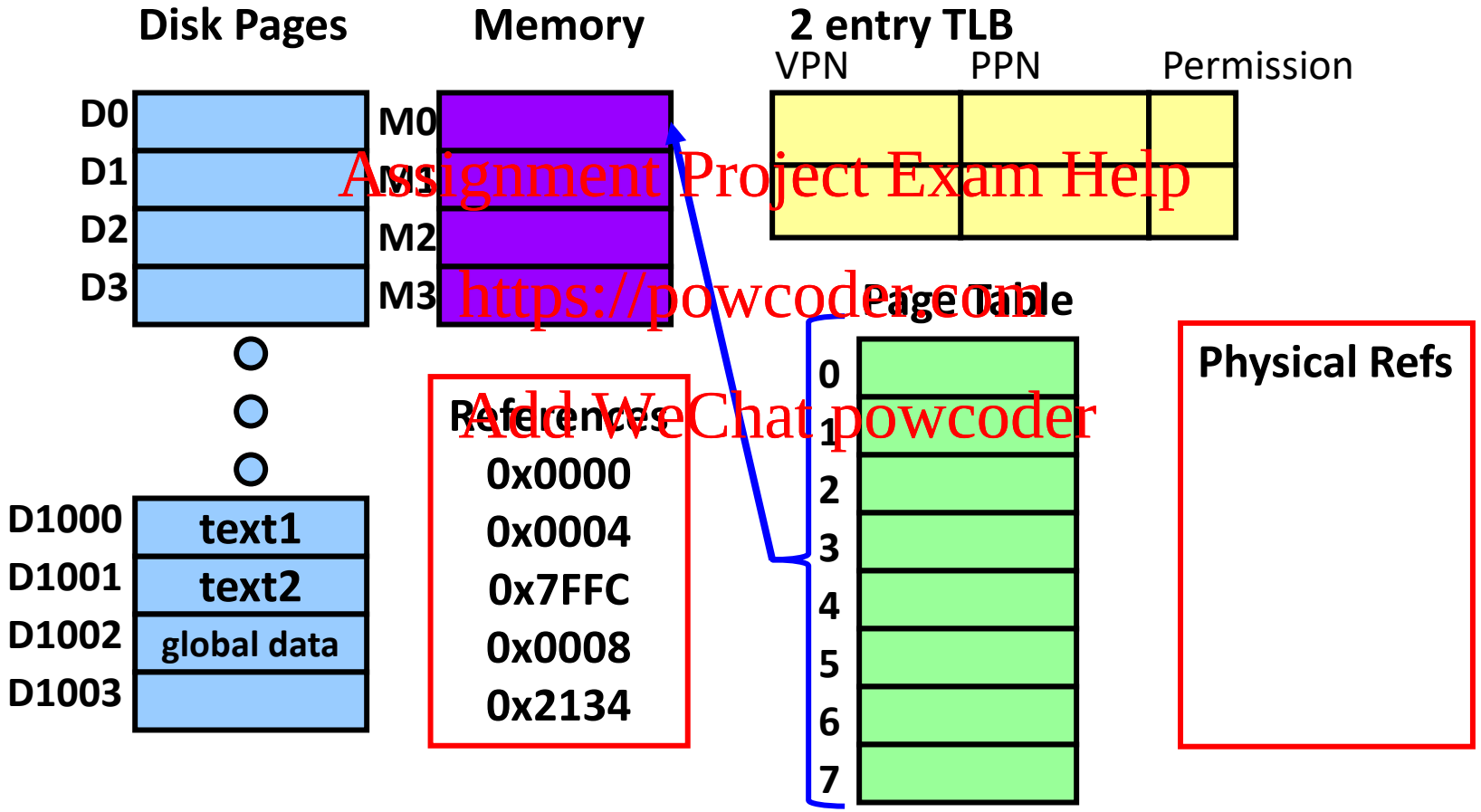
Loading a program into memory

Page size = 4 KB, Page table entry size = 4 B

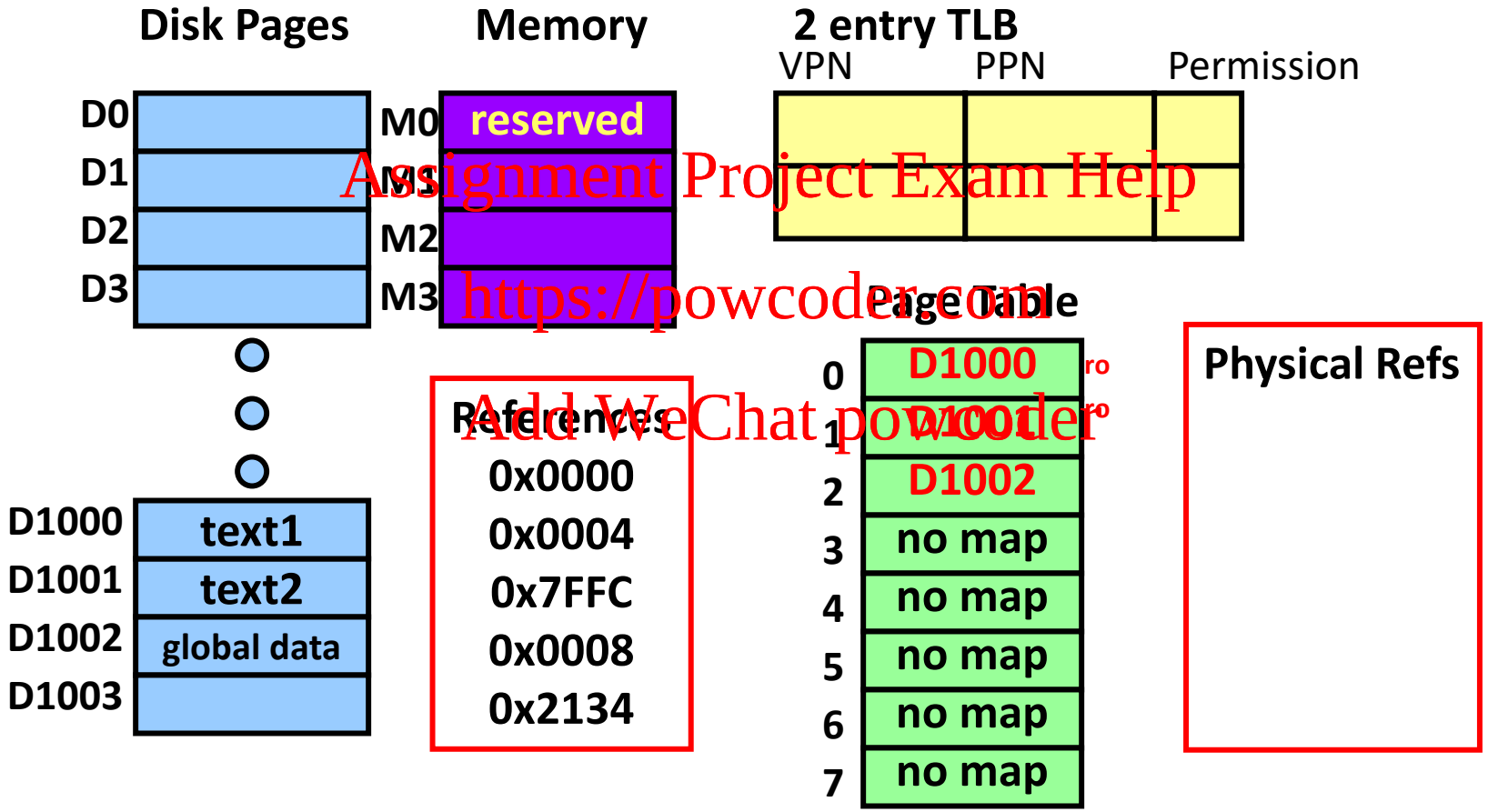
Page table register points to physical address 0x0000



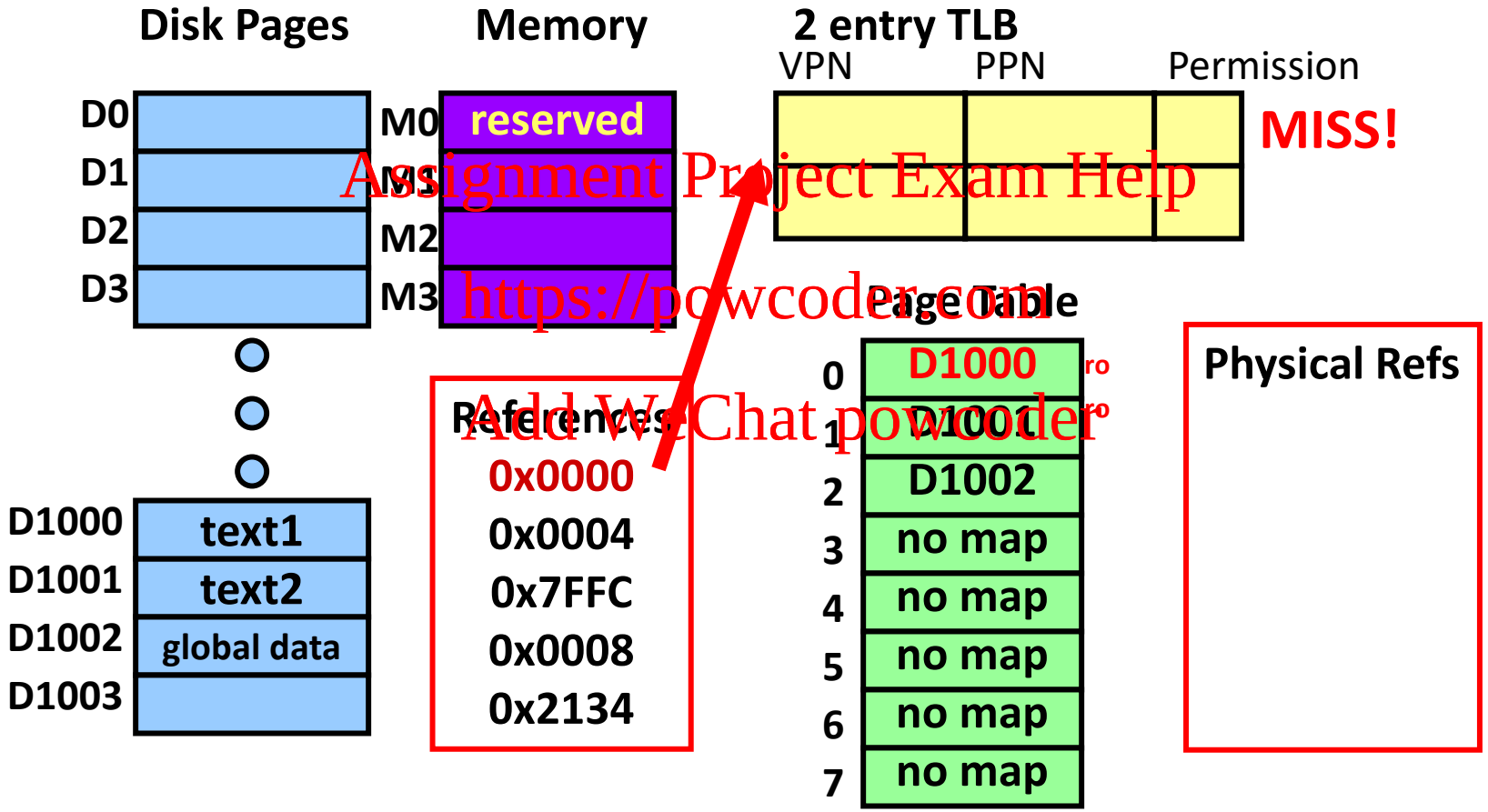
Loading a program into memory



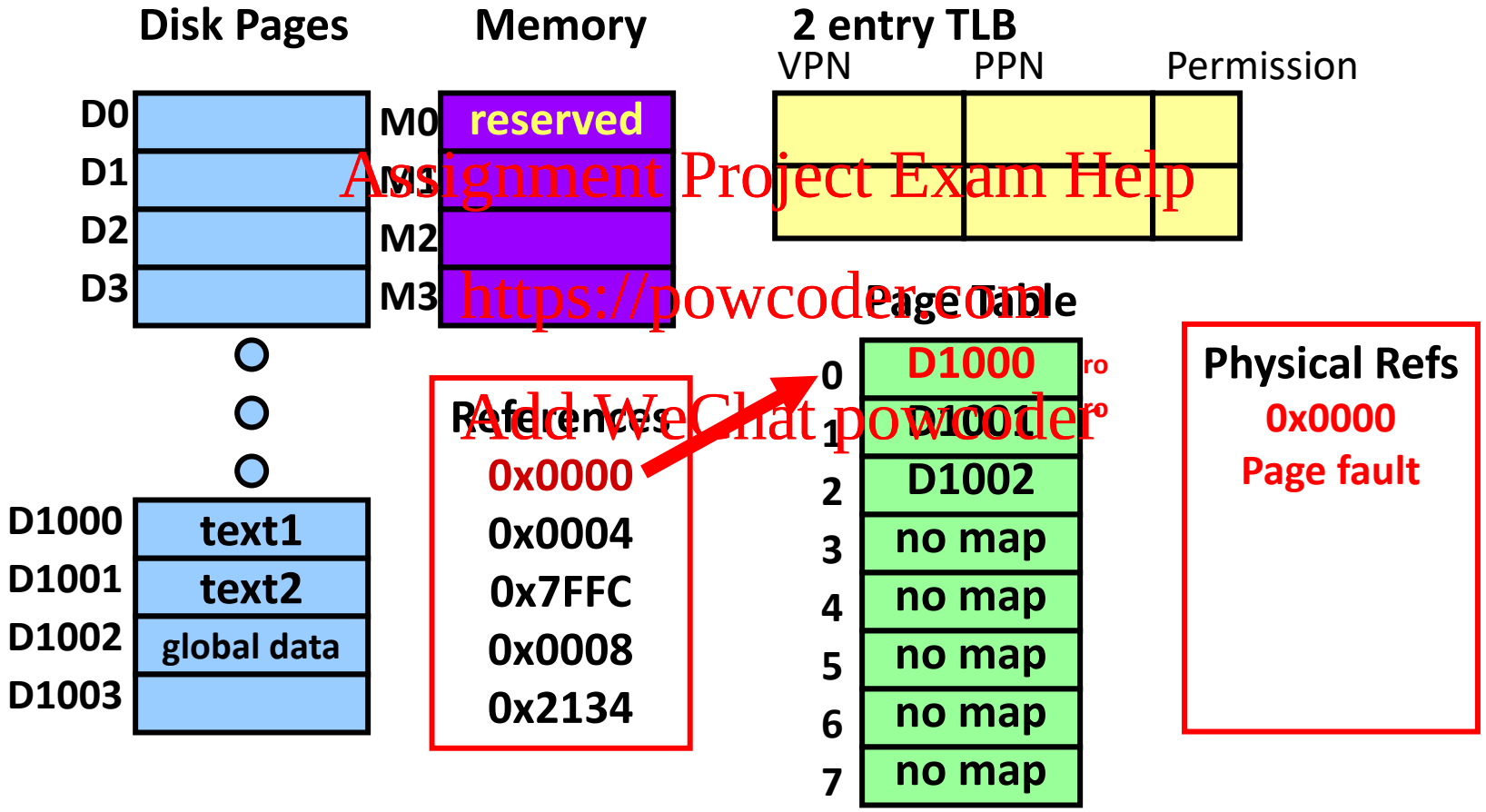
Step 1: Read executable header & initialize page table



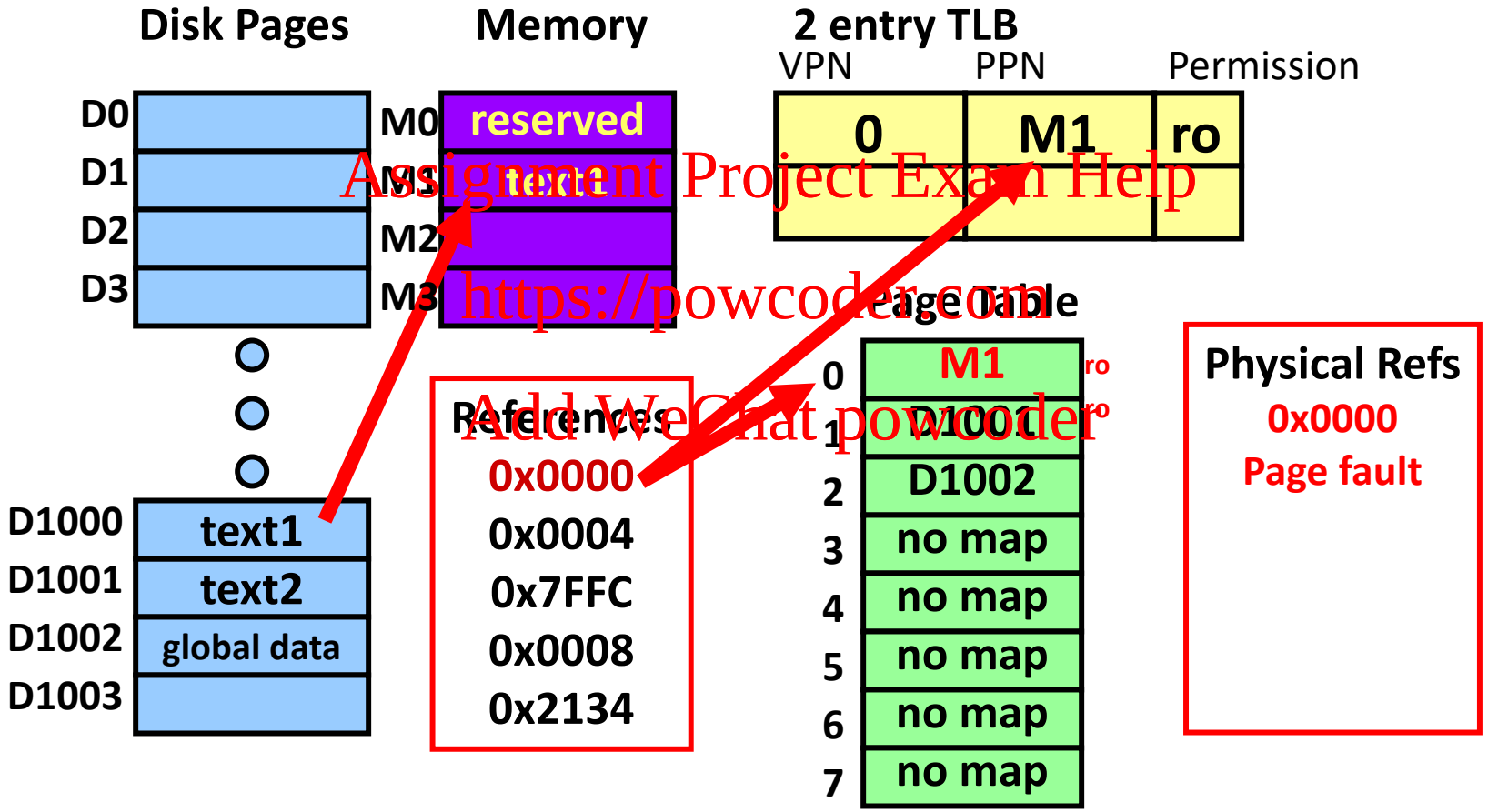
Step 2: Load PC from header & start execution



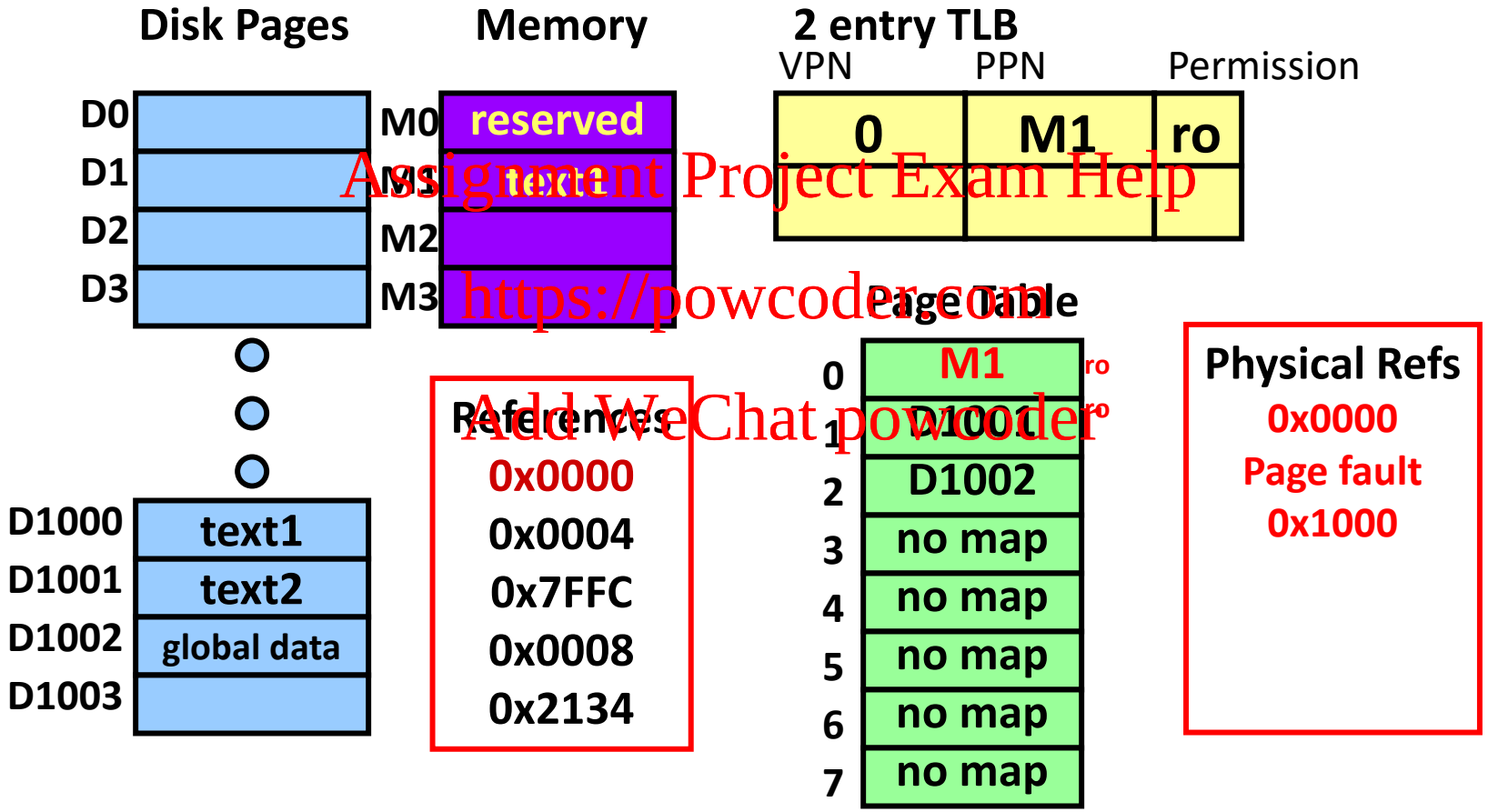
Fetching instruction 0000



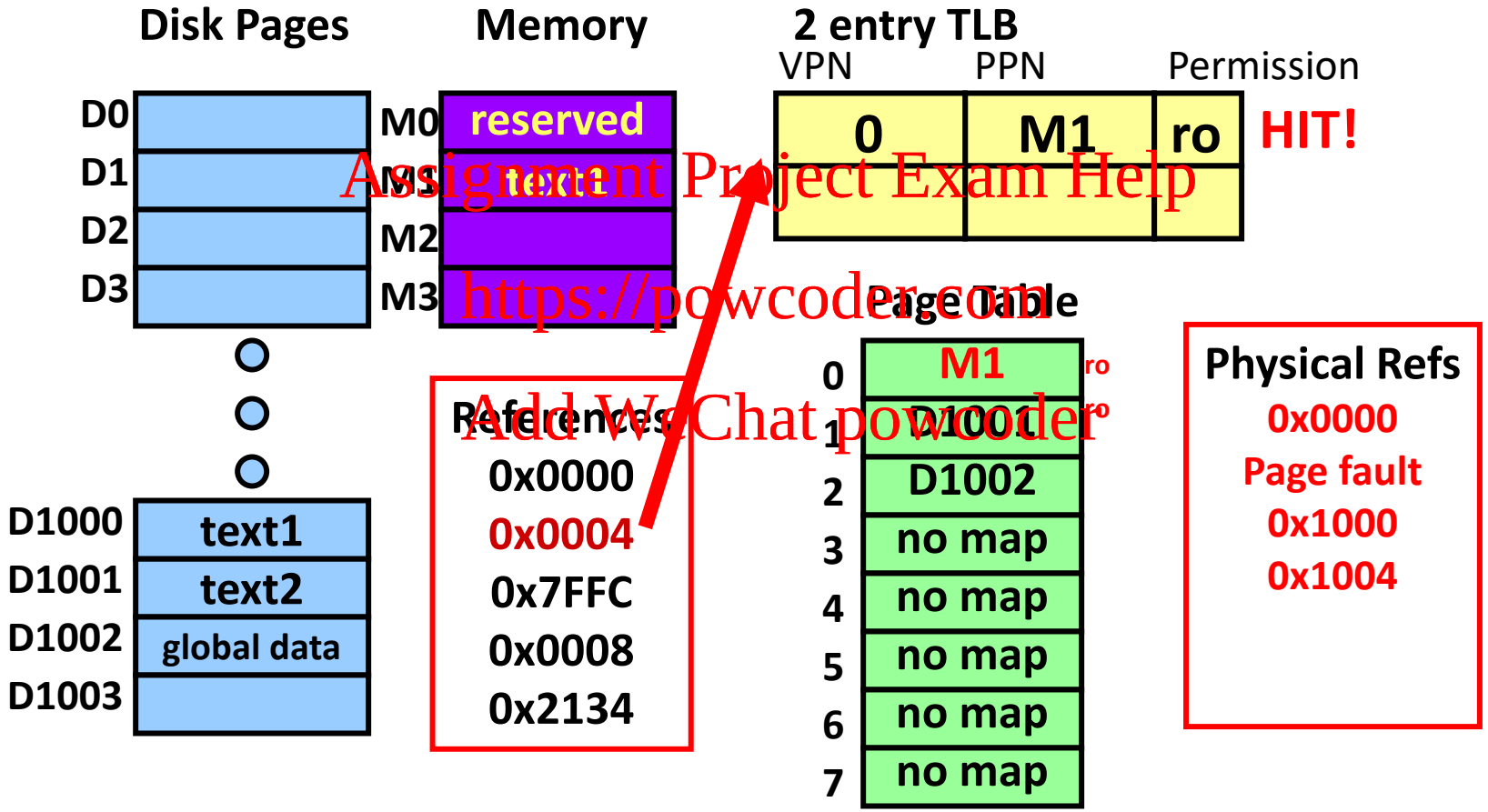
Fetching instruction 0000



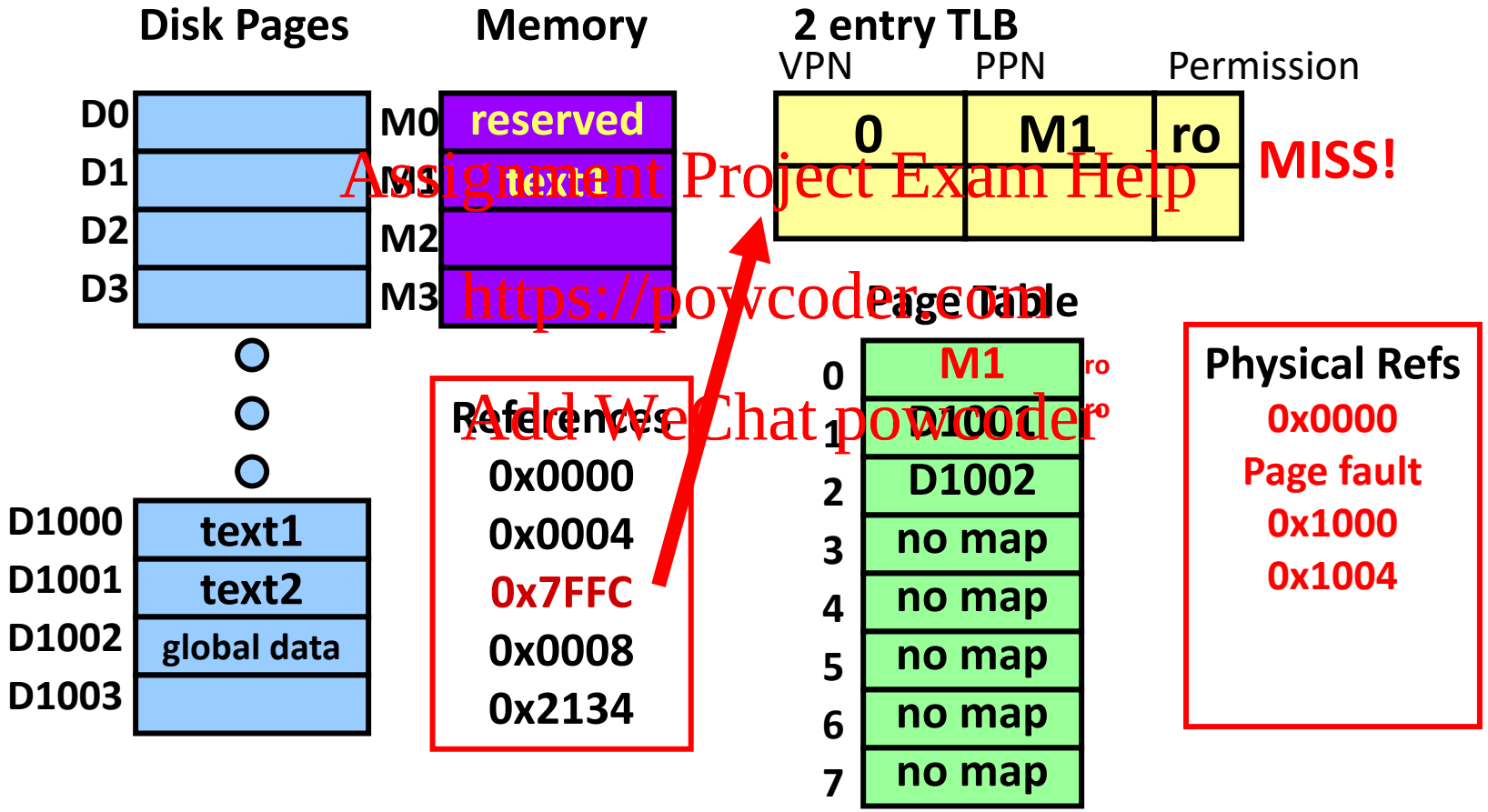
Fetching instruction 0000



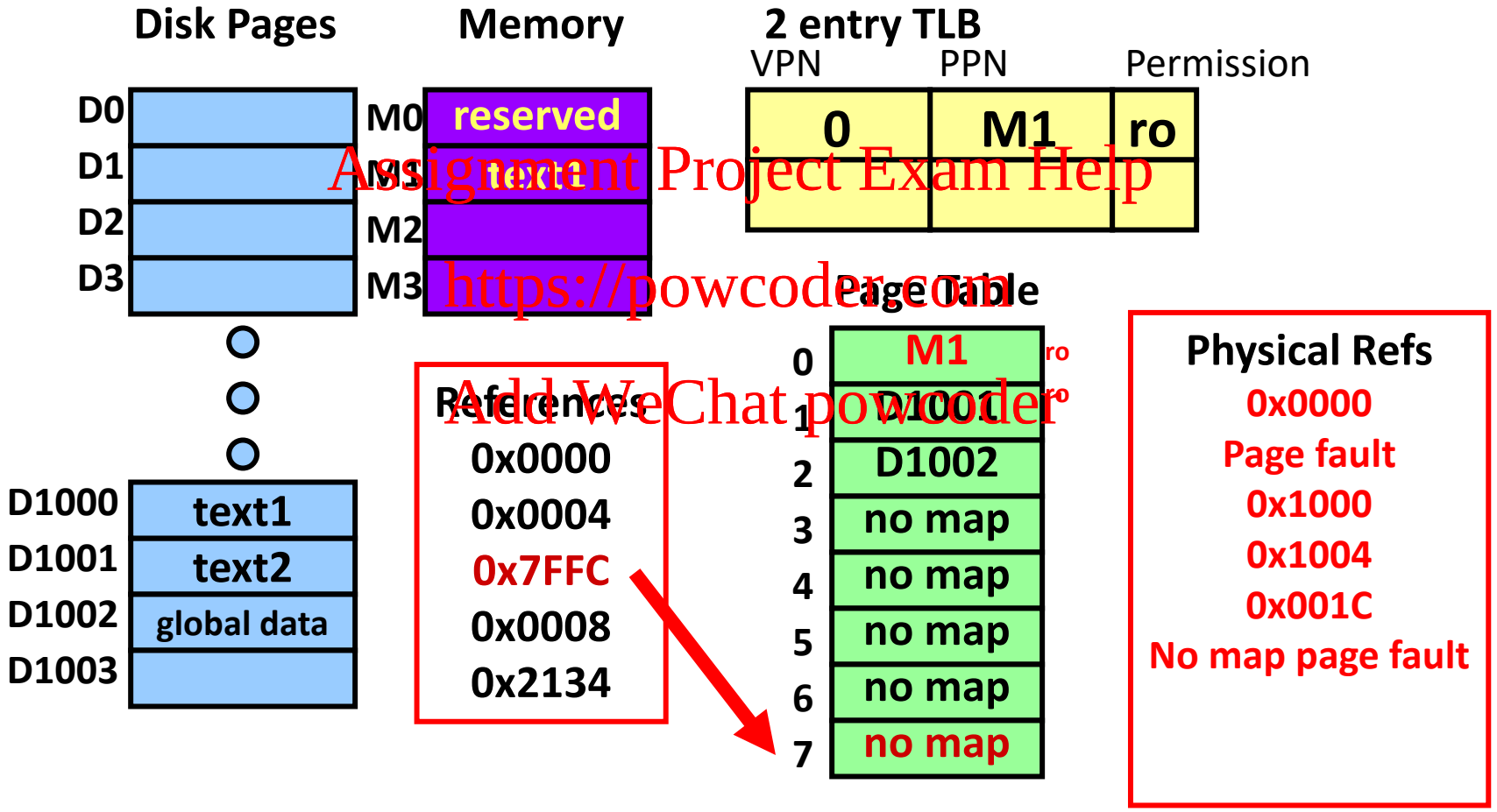
Fetching instruction 0004



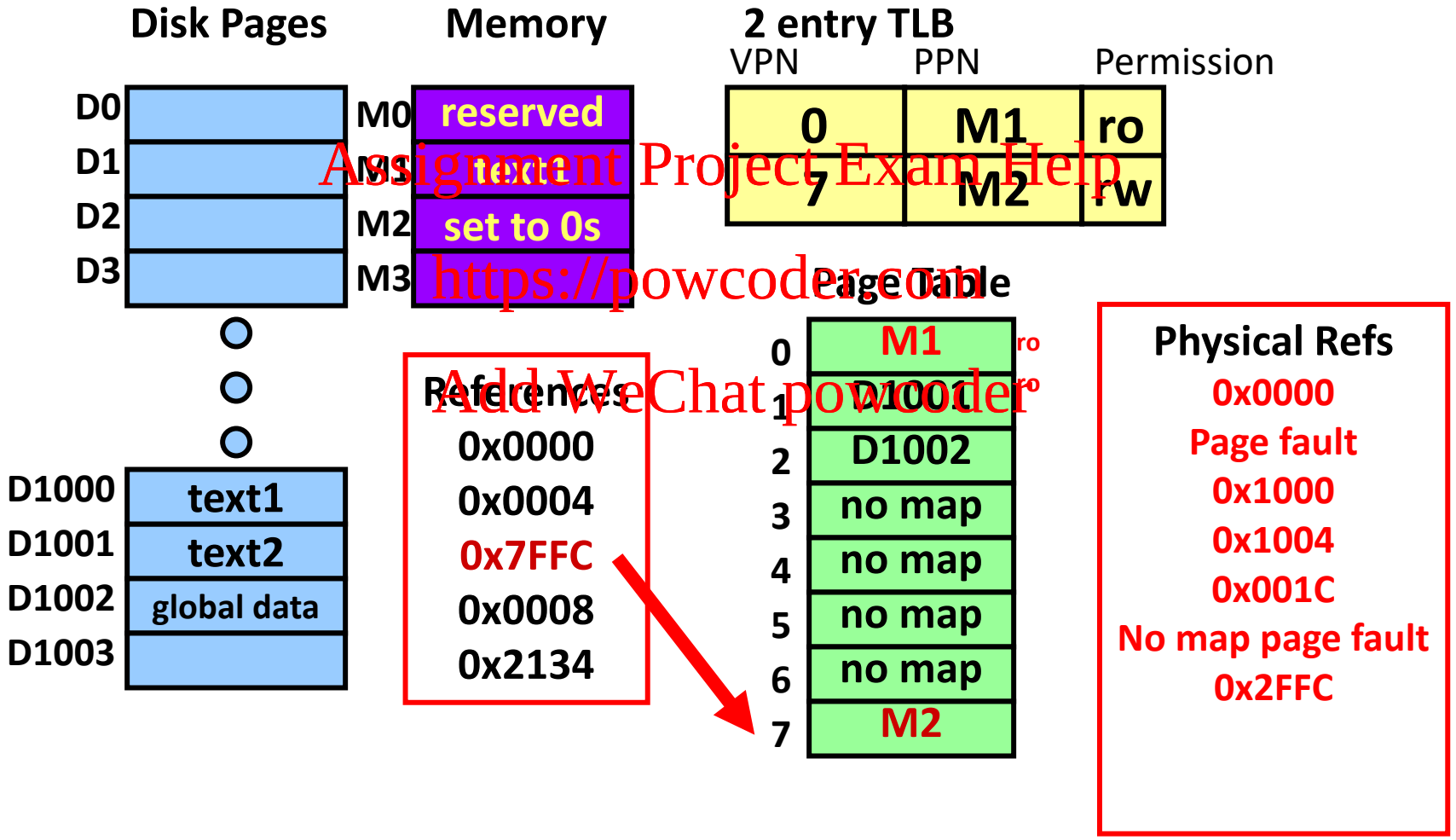
Reference 7FFC



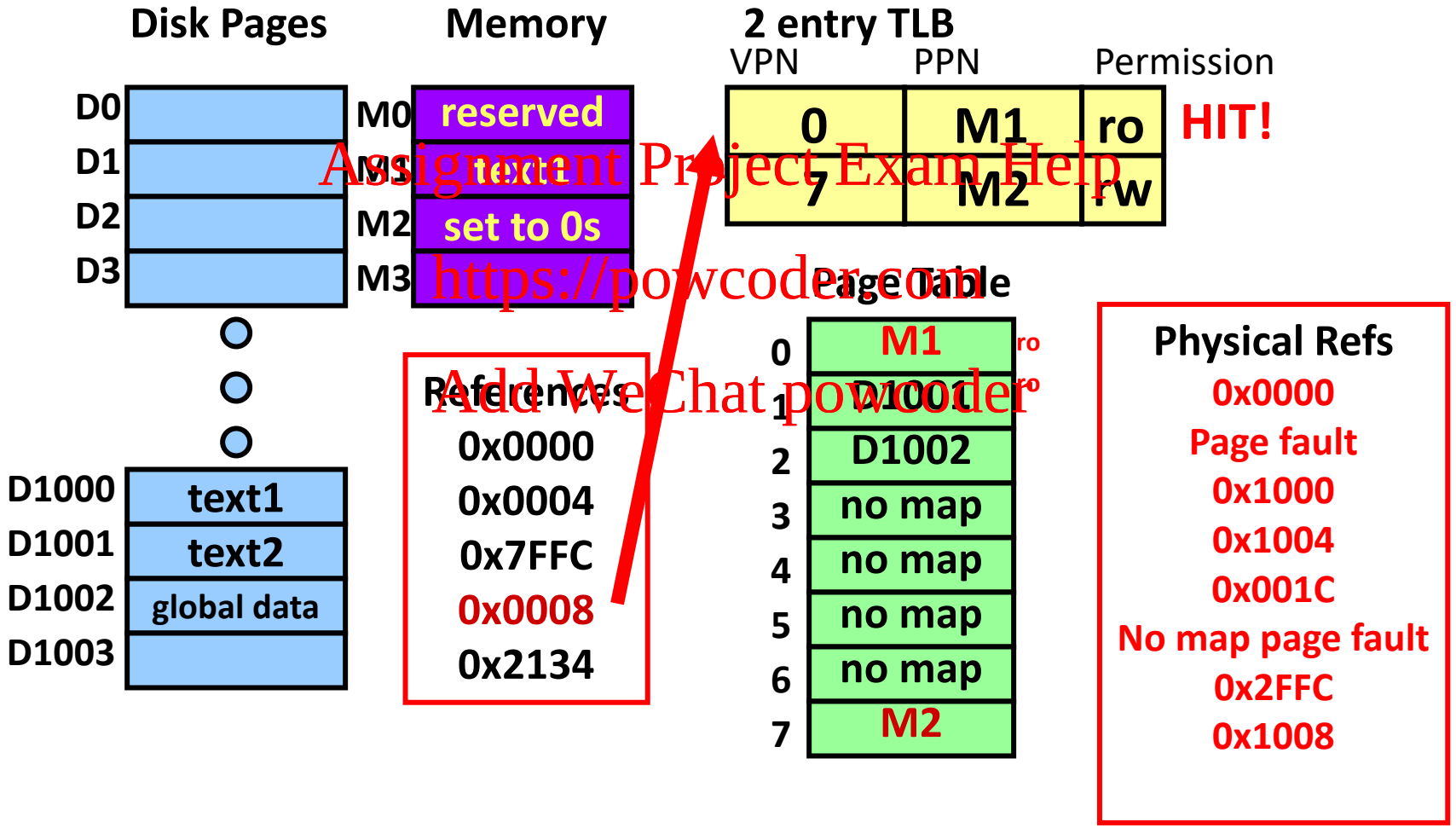
Reference 7FFC



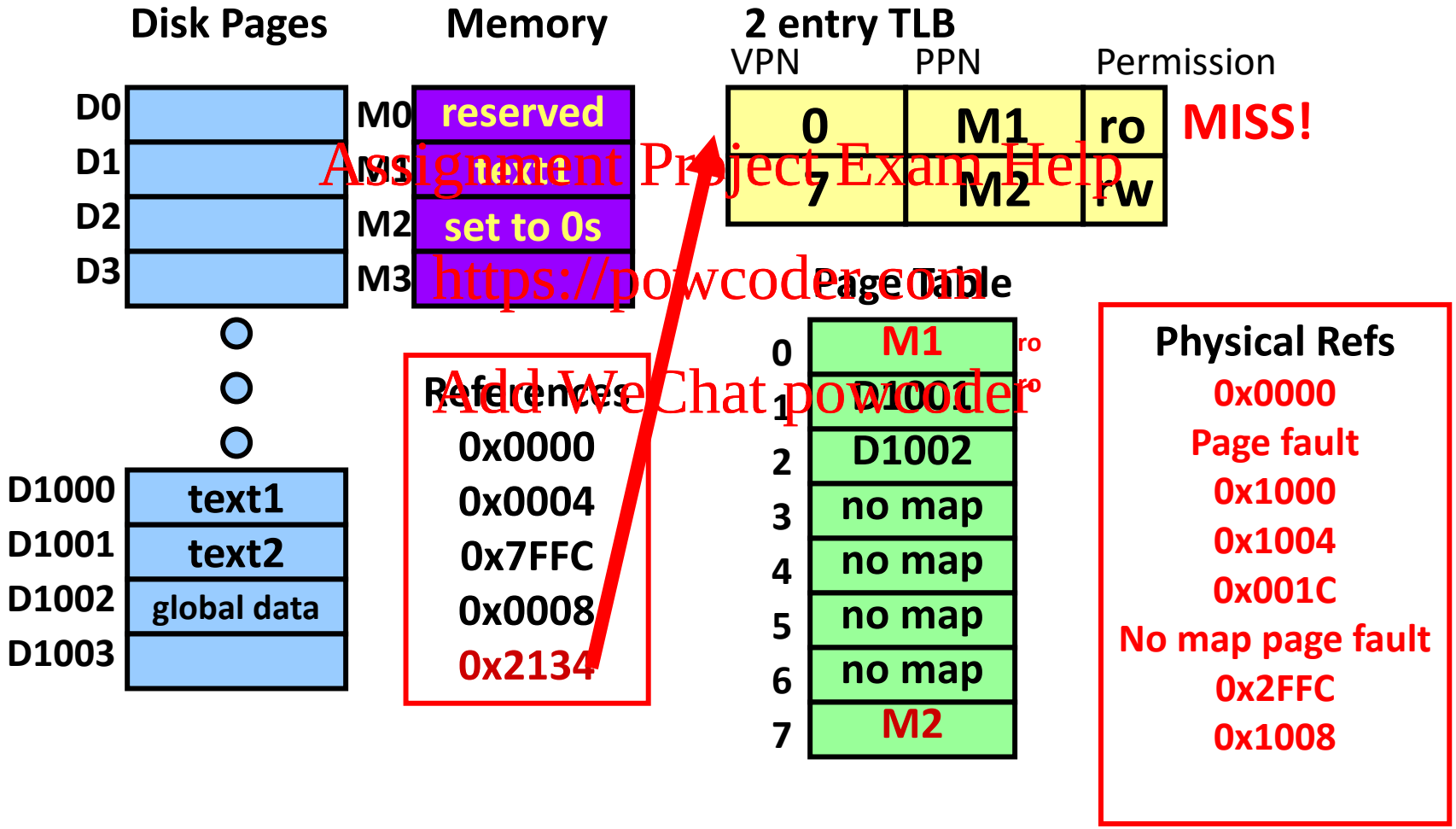
Reference 7FFC



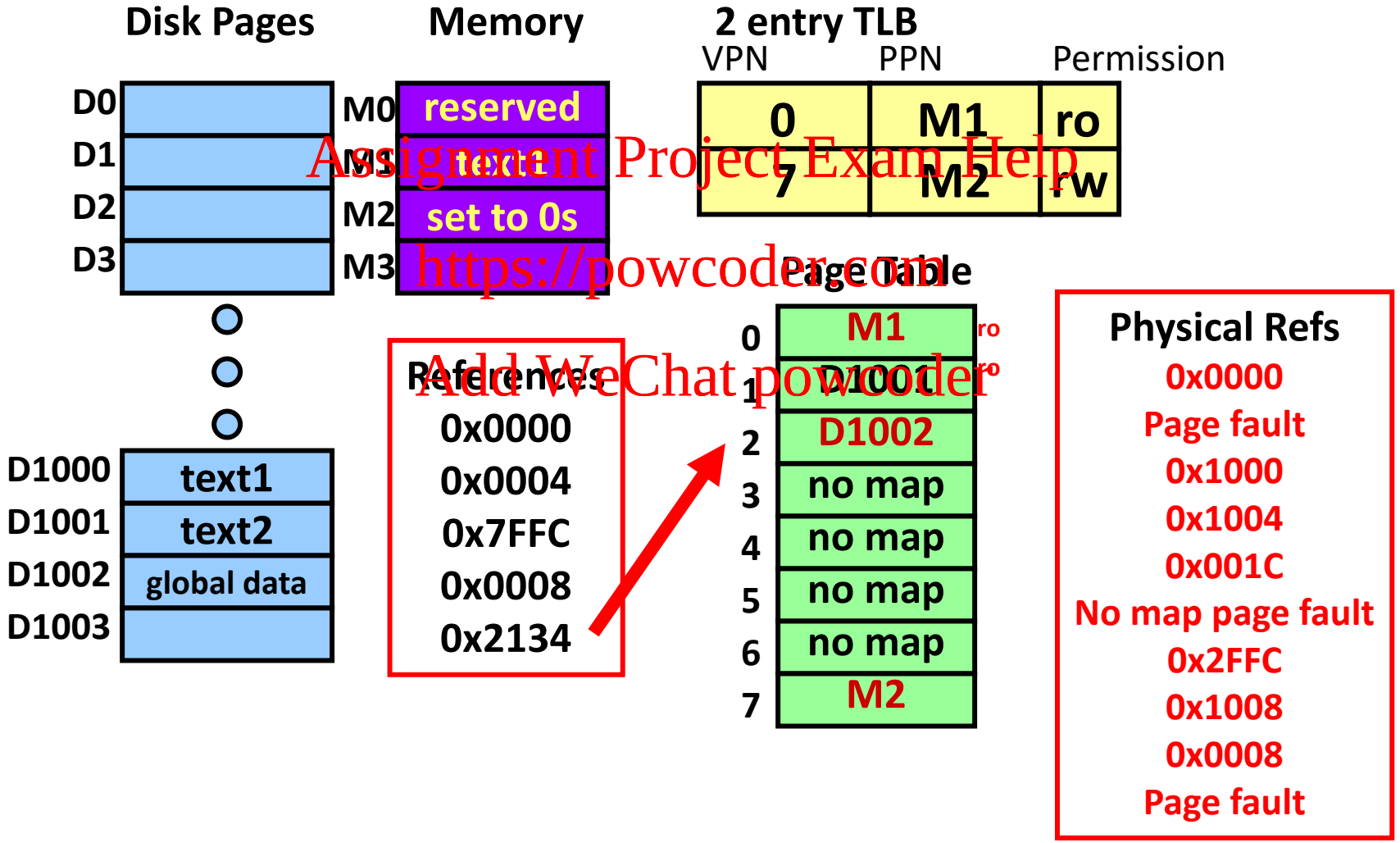
Fetching instruction 0008



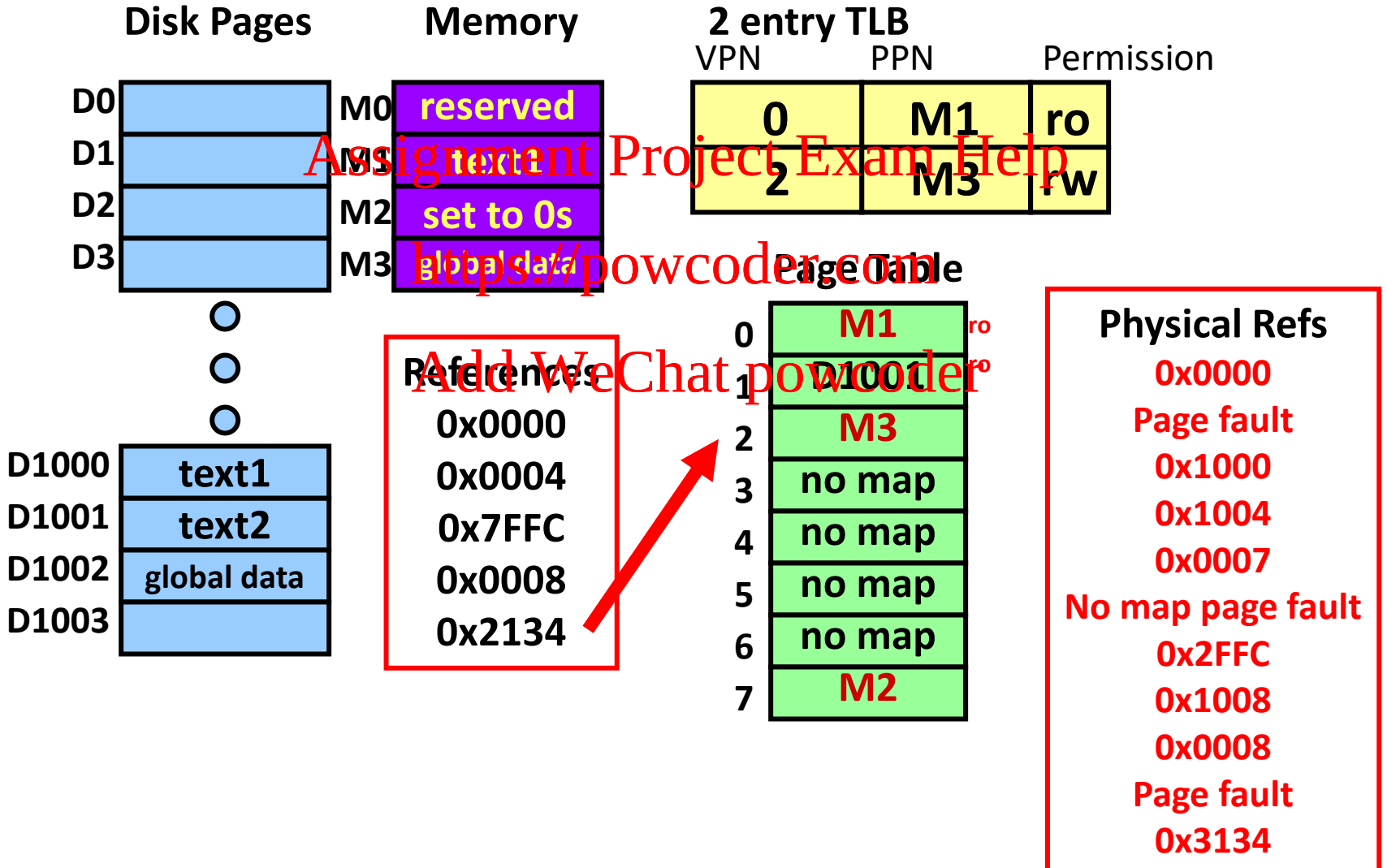
Reference 2134



Reference 2134



Reference 2134



Can we skip address translation?

Virtually-addressed caches

Address translation in a pipeline

Load/store: Memory stage

Instruction fetch: Fetch stage

When to do address translation?

After VA is computed, but before memory access is performed

Is it possible to skip address translation for some memory accesses?

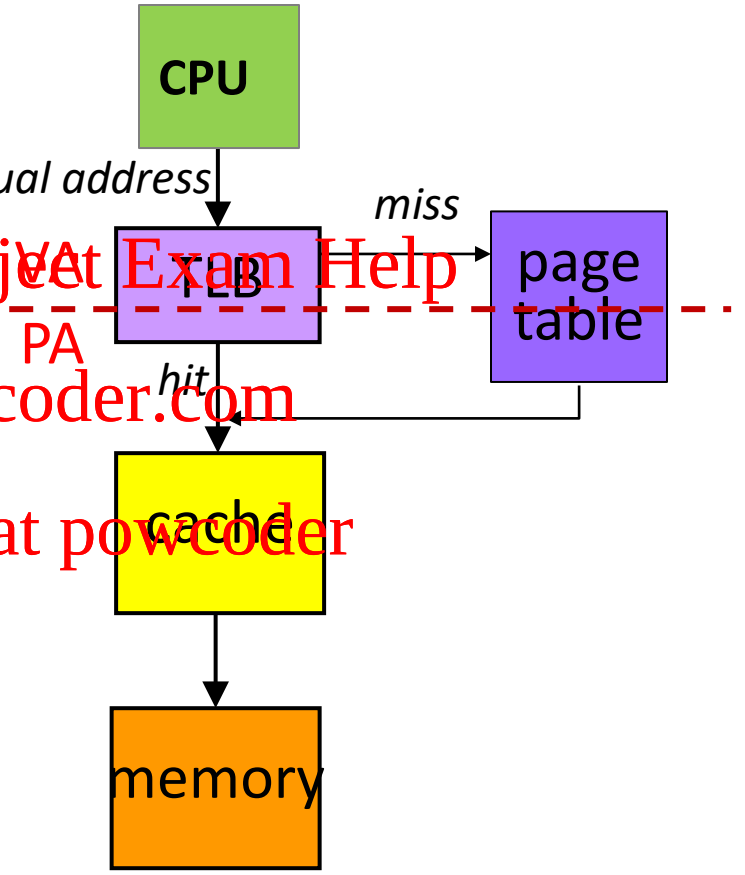
Yes. Answer: Virtually-addressed caches.

Option 1: Address translation *before* cache access

Physically-addressed Cache

Slower

Low complexity

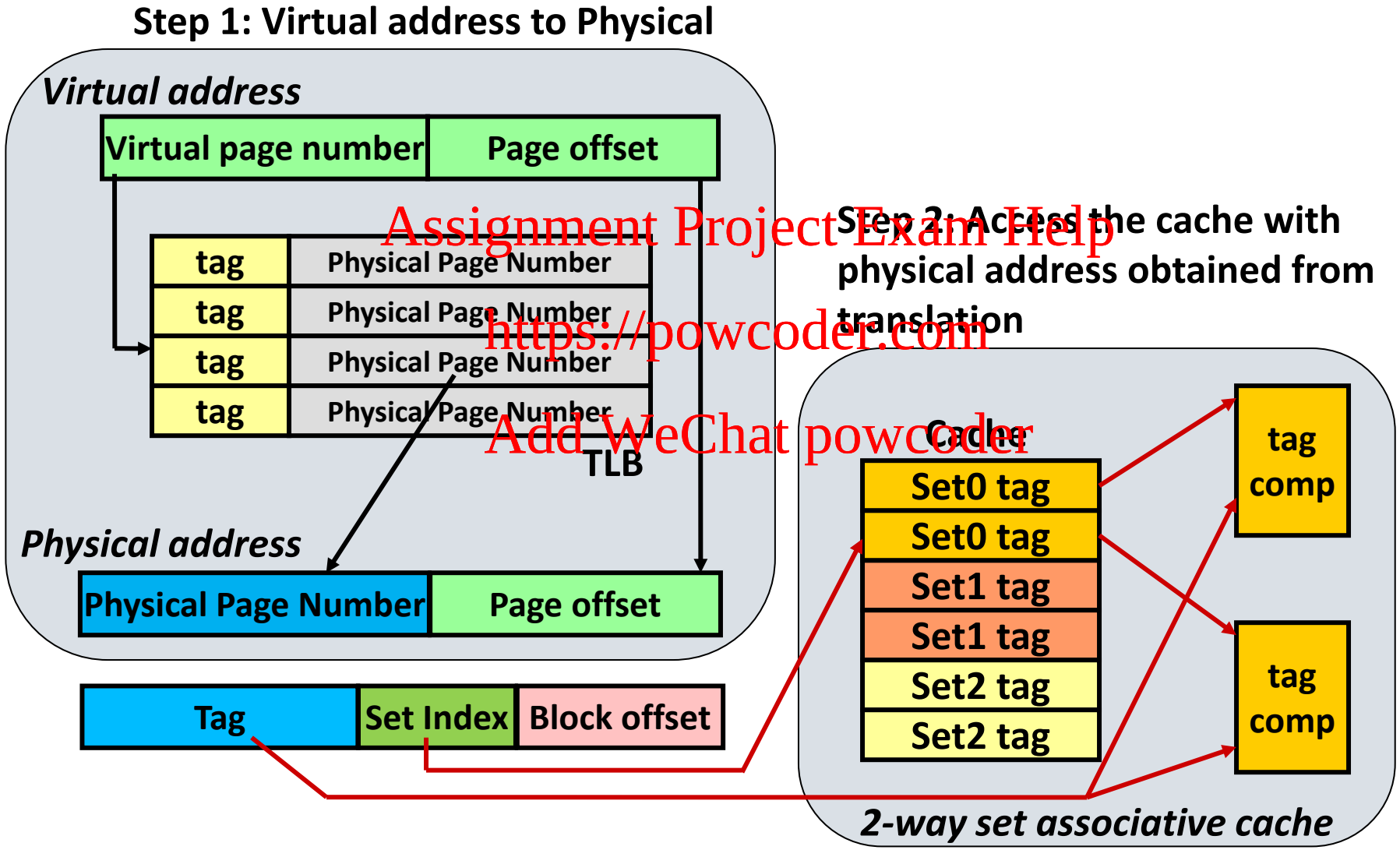


Use ***physical*** address to access cache

(tag, set index, and block offset bits)

Address translation only for all accesses

Physically addressed caches

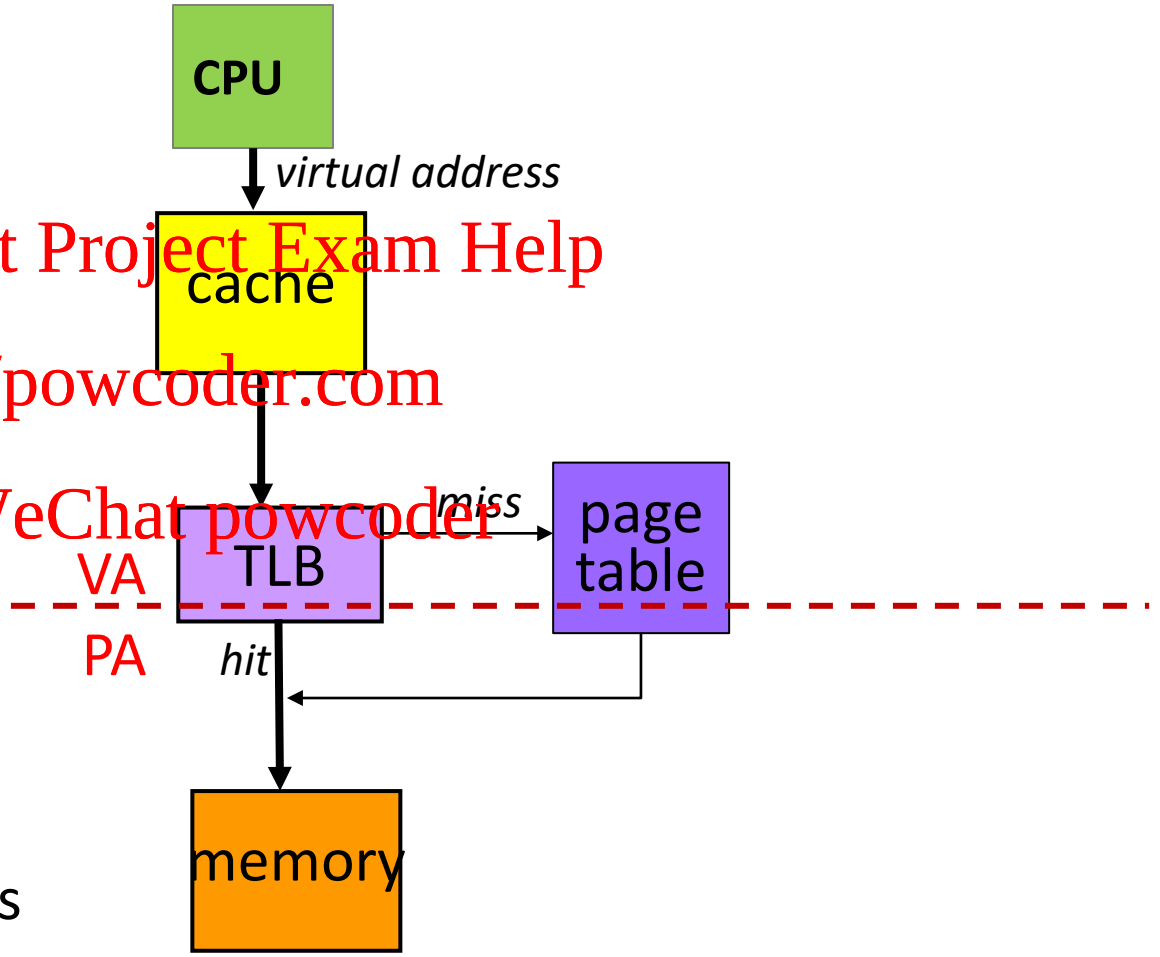


Option 2: Address translation *after* cache access

Virtually-addressed Cache

High complexity
(process isolation is hard)

Faster



Use ***virtual*** address to access cache
(tag, set index, and block offset bits)

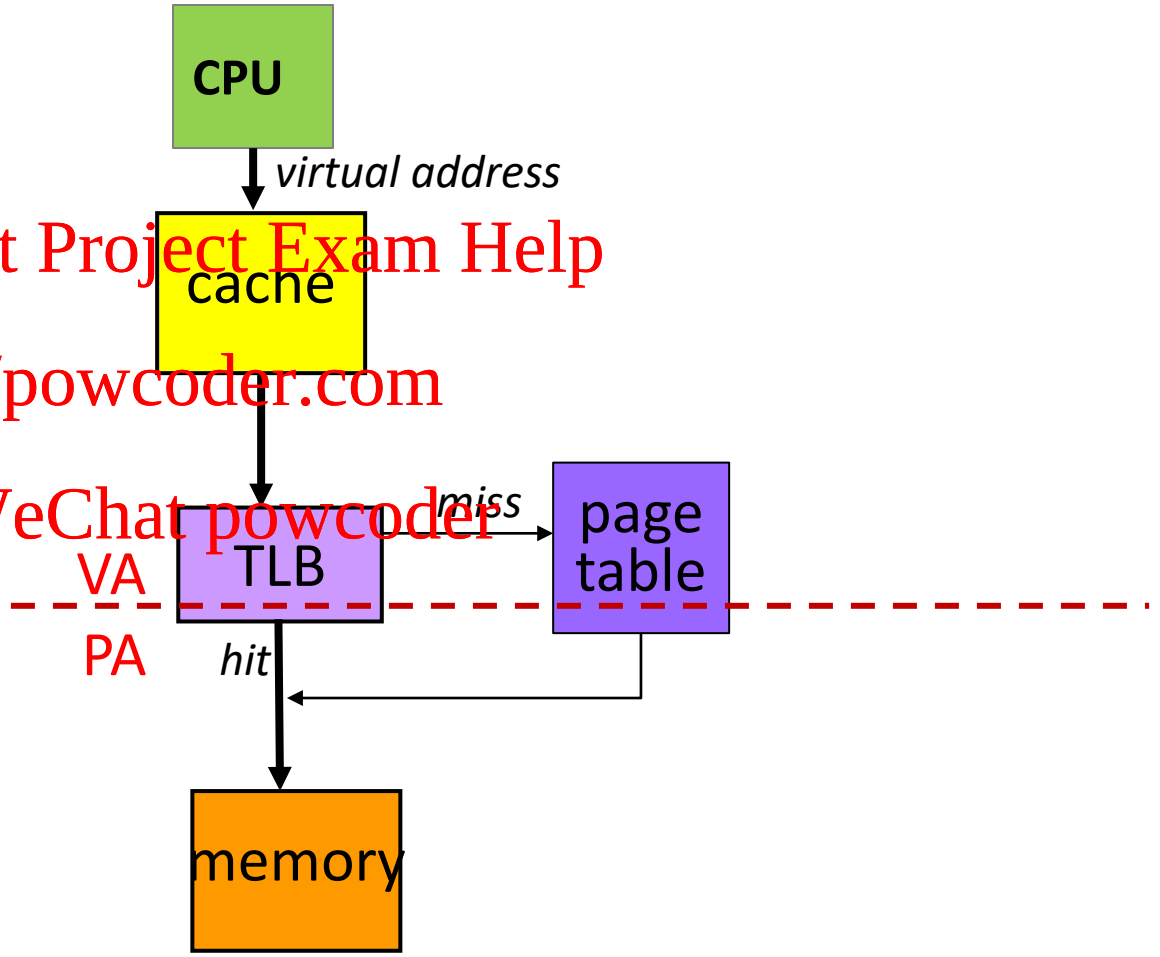
Address translation only for cache misses

Option 2: Address translation *after* cache access

Virtually-addressed Cache

High complexity
(process isolation is hard)

Faster

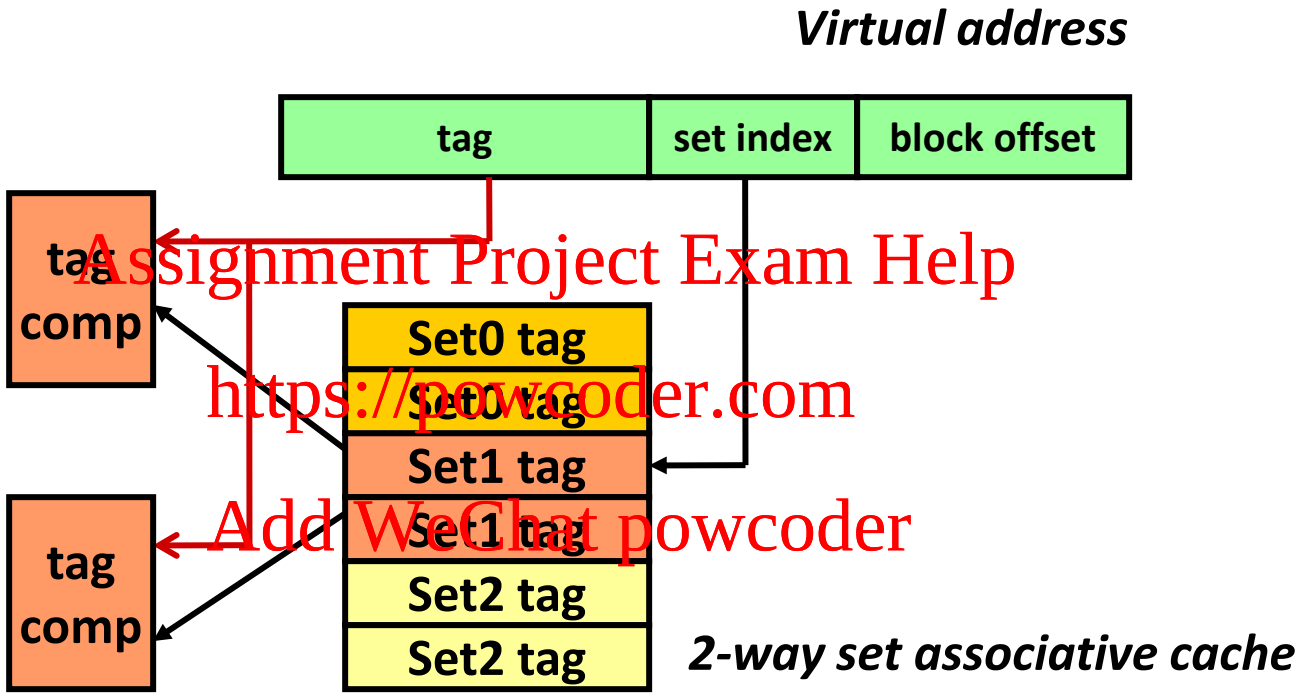


TLB can be accessed in parallel with cache lookup.

Saves time on a cache miss, where physical address is needed to access main memory

But wasteful TLB accesses on cache hits

Virtually addressed caches



Address translation: Before or After Cache Access

Physically-addressed cache

Generate virtual address ->

Access TLB ->

Access cache ->

if cache miss, access main memory

Virtually-addressed cache

Generate virtual address ->

Access cache ->

if cache miss, access TLB ->

Access main memory

Before Cache

After Cache

Tradeoffs

Physically-addressed caches: **Slow; Simple**

Vs

Virtually-addressed caches: **Fast; Complex**

Physical Vs Virtual Caches: Latency

Physically-addressed caches

Cache is accessed with physical address (after VM translations).

- Slow. Cache can be accessed only after address translation
- Inefficient, because all accesses need address translation

Virtually-addressed caches

Cache is accessed with virtual address (before VM translation).

- Fast. Skips address translation for cache hits.
- Efficient, because only cache misses need address translation

Physical Vs Virtual Caches: Complexity

Problem for virtually-addressed cache:

The same virtual address refers to different physical addresses in two different processes.

To ensure process isolation, on a context switch:

Virtual cache need to be invalidated. Dirty cache blocks written back.

Physical cache need not be invalidated.

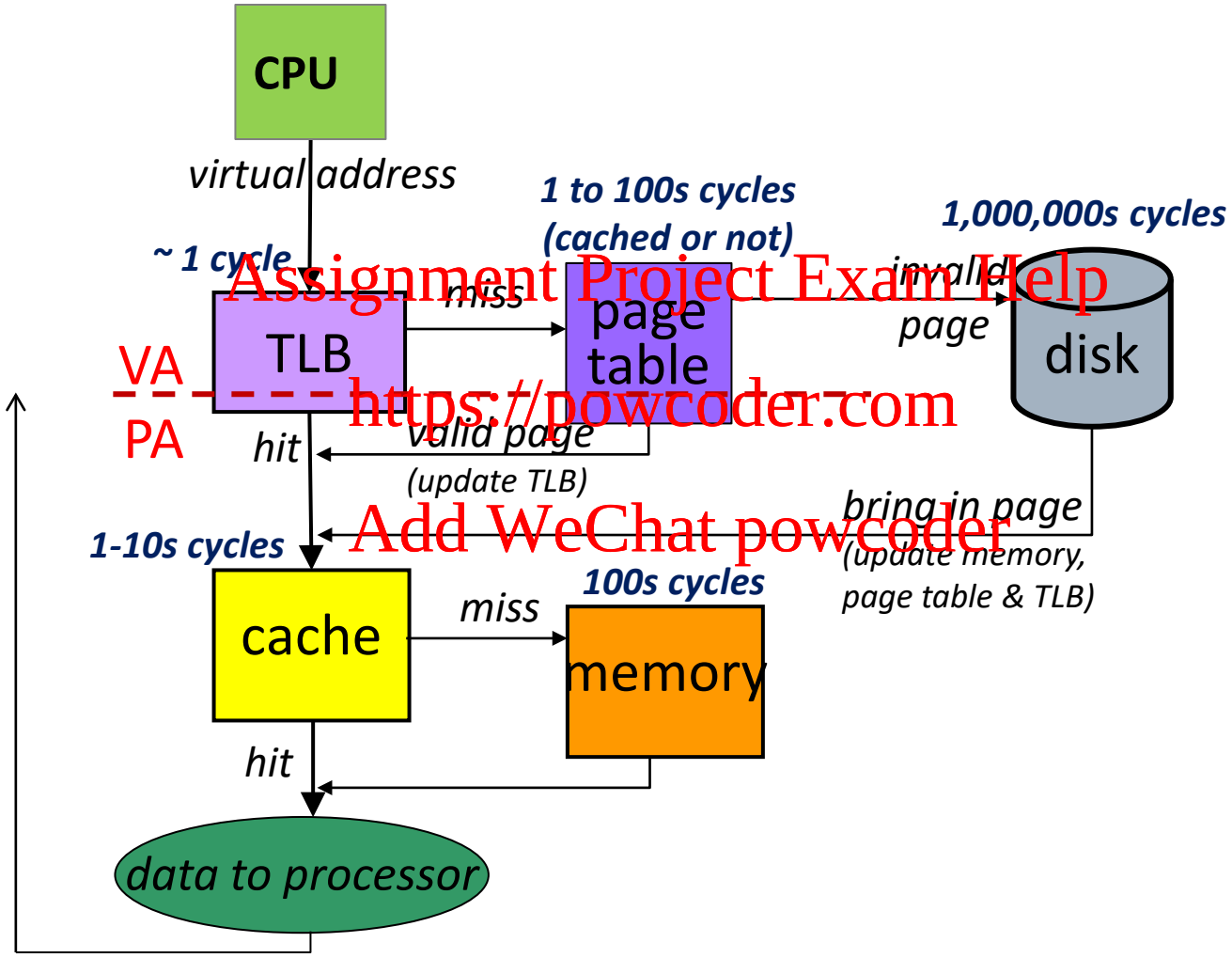
So, physical cache incurs fewer cache misses if context switches are very frequent (but generally they are not)

Typically, in modern processors

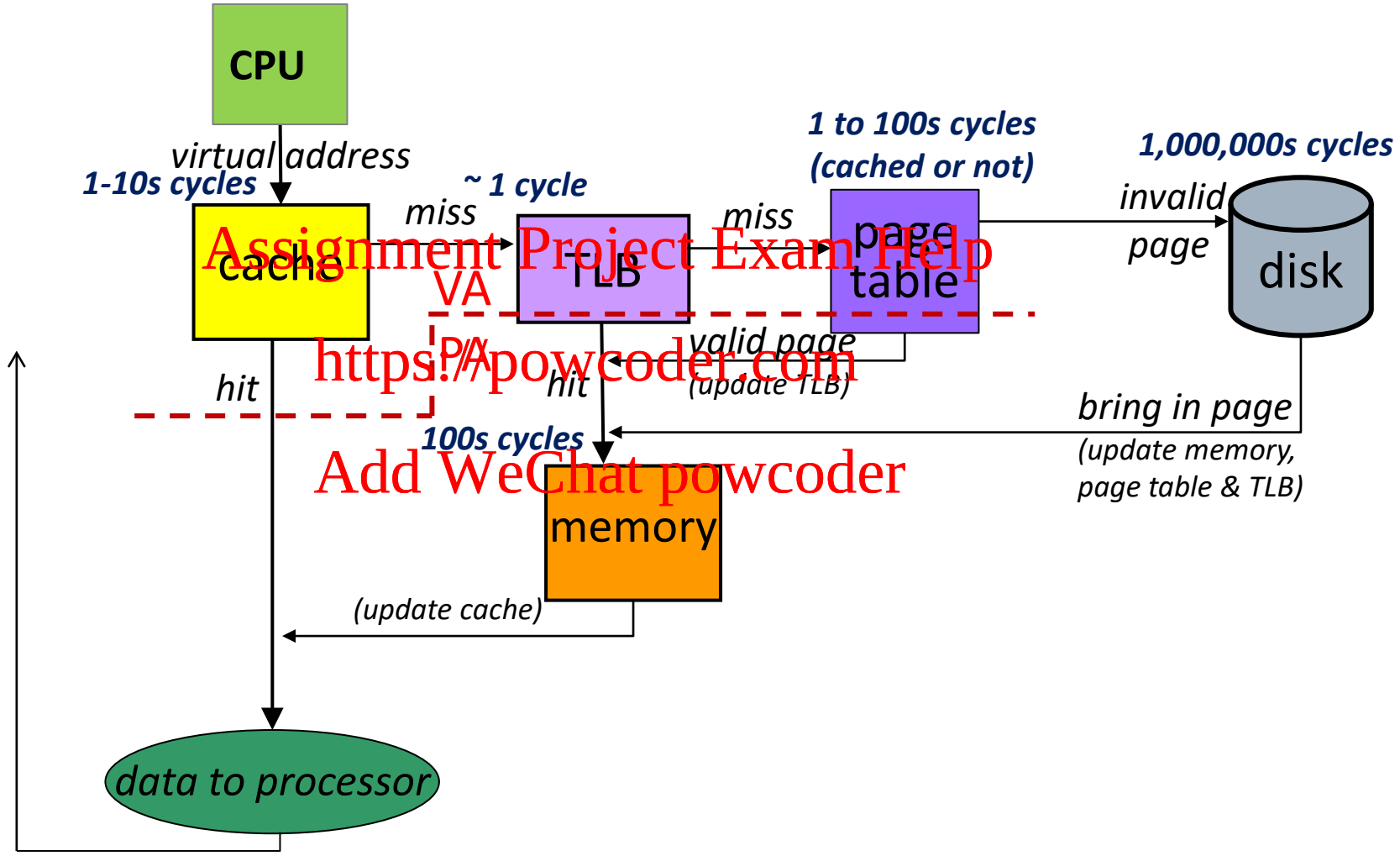
Level-1 (L1) caches are virtually-addressed. L1 needs to be fast.

L2 and L3 are physically-addressed. Larger structures.
Checking TLB before accessing them does not significantly affect their latency.

Physically addressed caches: detailed flow



Virtually addressed caches: detailed flow



OS Support for Virtual Memory

OS must be able to modify the page table register, update page table values, etc.

To enable the OS to do this, **BUT** not the user program, we have different execution modes for a process:

- Executive (or supervisor or kernel level) permissions and
- User level permissions.

References (not part of Course Syllabus)

See how Intel's memory management hardware works, Intel x86 Software Manual:
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

Chapter 4 is on Paging

Linux page table management:
https://www.kernel.org/doc/gorman/html/understand/un`derstand006.html