# Virtual memory – II
# TLB and Caches

Acknowledgements:

CO and design- Chapter 5 - Henessey and Patterson

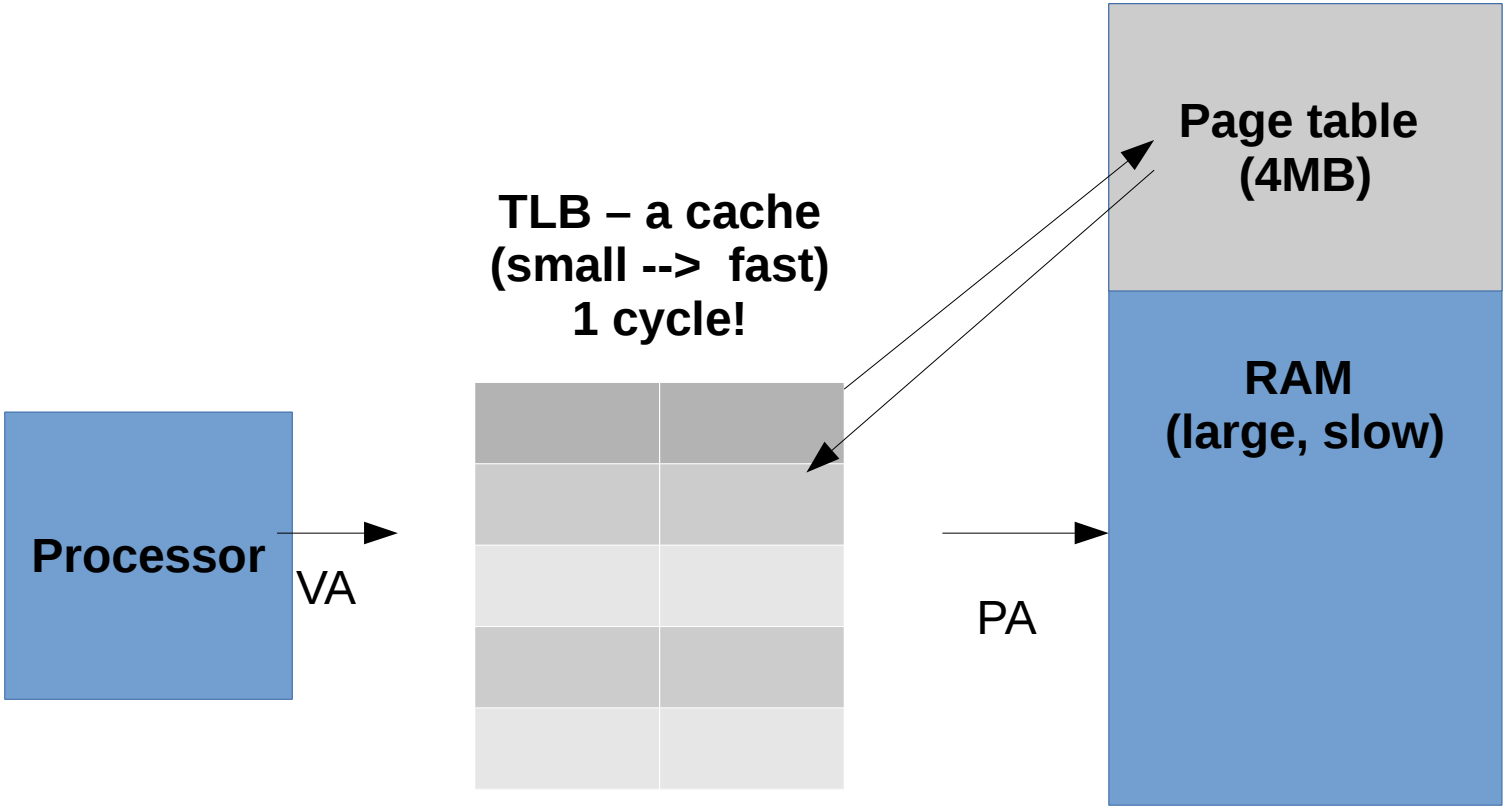
CA- Quant approach- H & P

HPCA- Georgia Tech- Virtual memory
https://www.youtube.com/watch?v=Dz9Hgq65iJw&list=PLAwxTw4SYaPn79fsplIuZG34KwbkYSedj

# Make VM fast

- For each memory access:
  - Access Page table **in RAM**
  - Translate
  - Access data **in RAM**
  - **2 memory accesses**
- **PT Look up fast?**

# Translation look aside buffer (TLB)




- **<u>TLB – Cache of recent virtual to physical address translations.</u>**
- **<u>Stores the final frame number (even in case of a multi-level PT)</u>**
- iTLB and dTLB
- Small size, fully or highly associative
- Eg - TLB: 64 entries  (for 4kB pages) or 512 entries
- Can have a 2- level TLB

# Size

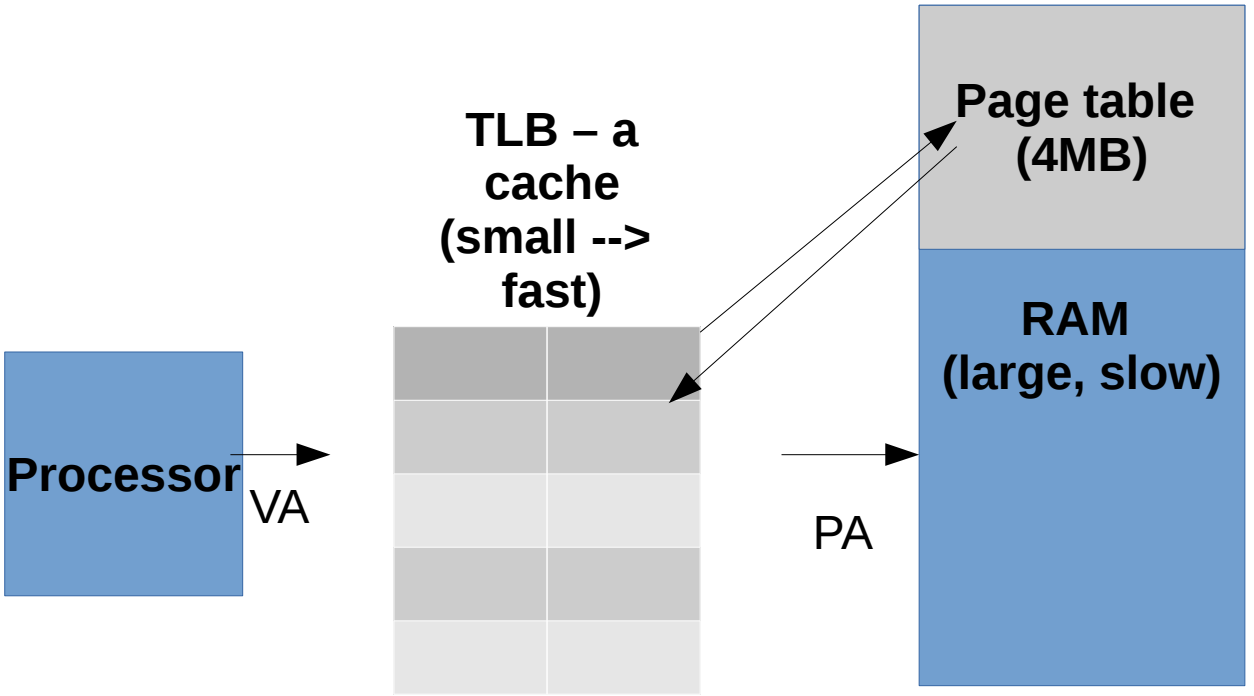
For a 32 bit program, there are 1million entries in Page Table

Assume page size of 4kB and TLB has 64 entries

--> Each TLB entry covers 4kB pages

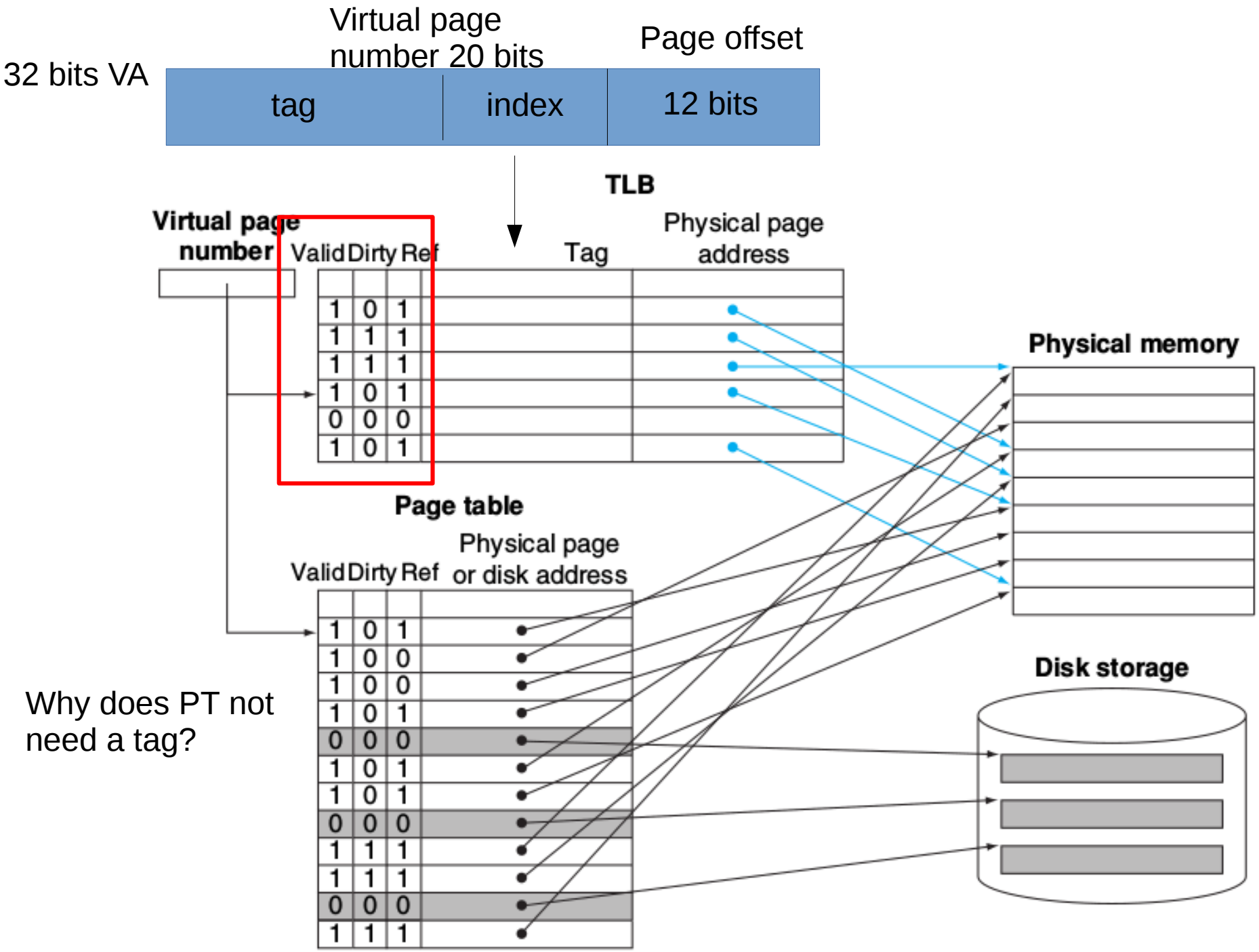
64 4kB pages --> 256kB

# Translation look aside buffer (TLB)

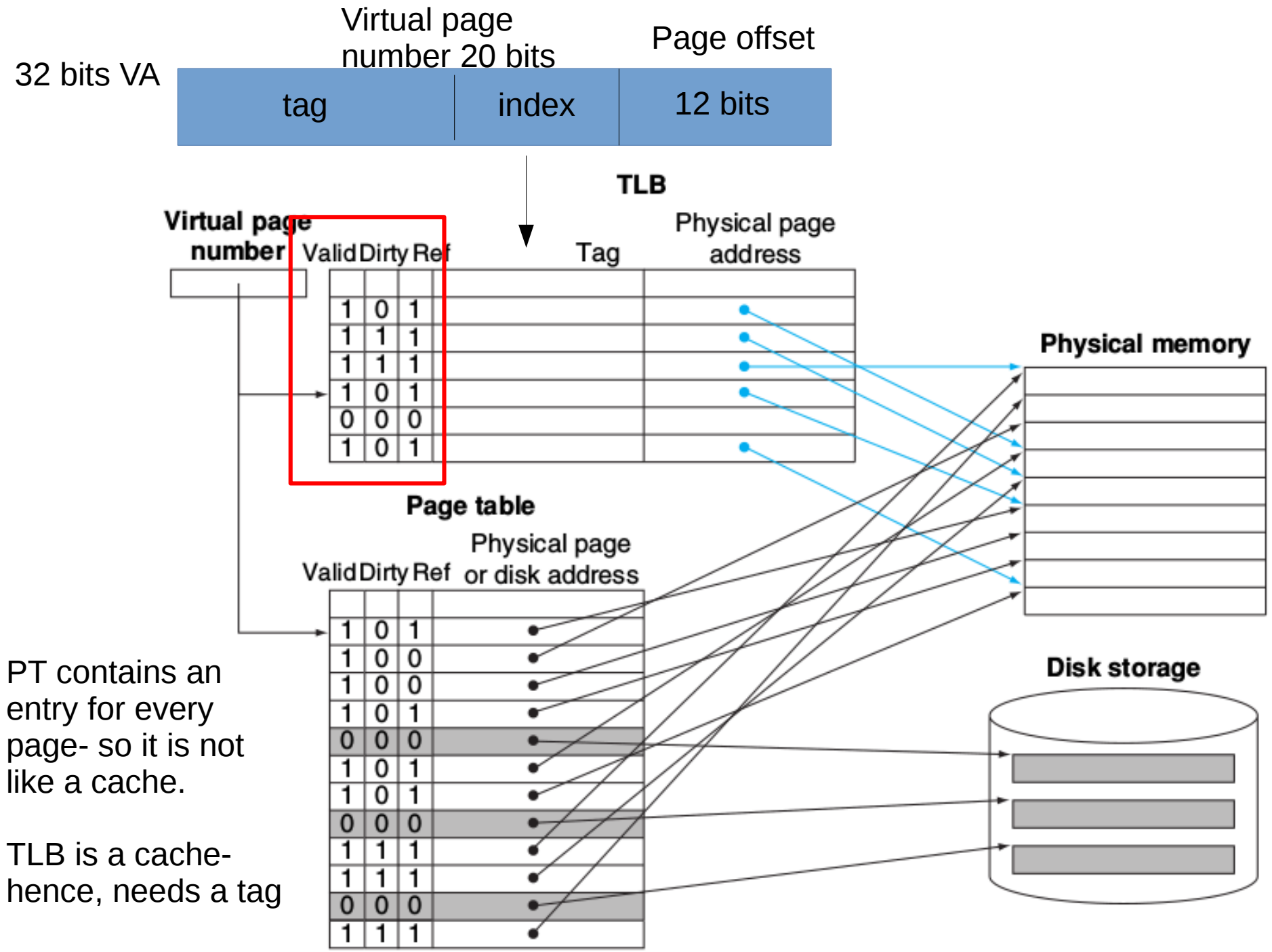



- Entry not in TLB -->
  - Go to PT and fetch the translation, go to RAM, fetch the page/data - SLOW
  - Go to PT and fetch the translation –> Page not in RAM, Go to disk and fetch, update PTE --> WORST
- Entry in TLB -->
  - Go to RAM, fetch the page/data – BEST
  - Page not in RAM? Unlikely, TLB entry will get evicted soon

# How does the TLB work




Why does PT not need a tag?

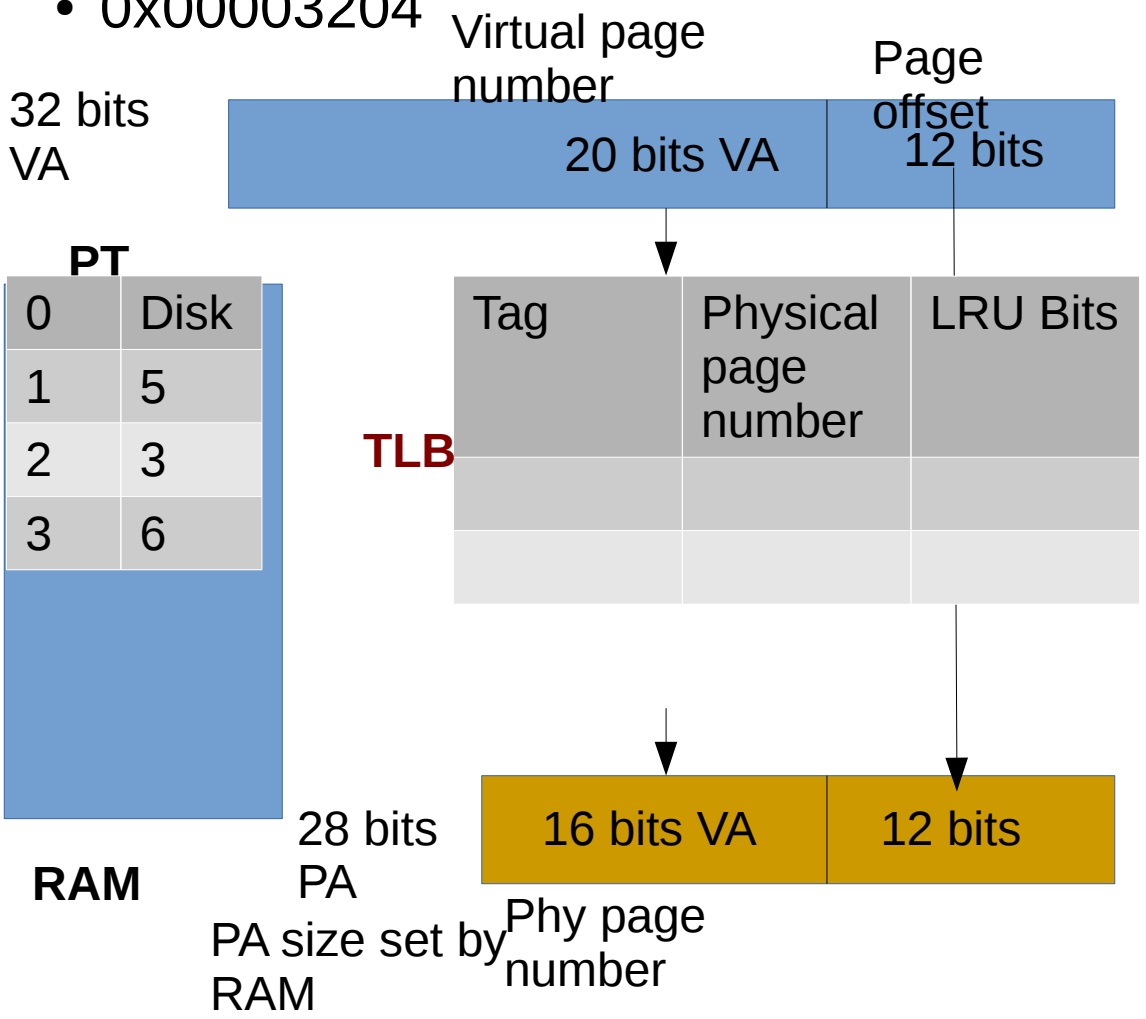



PT contains an entry for every page- so it is not like a cache.

TLB is a cache- hence, needs a tag

# Translation using TLB- assume Fully associative TLB (no index)

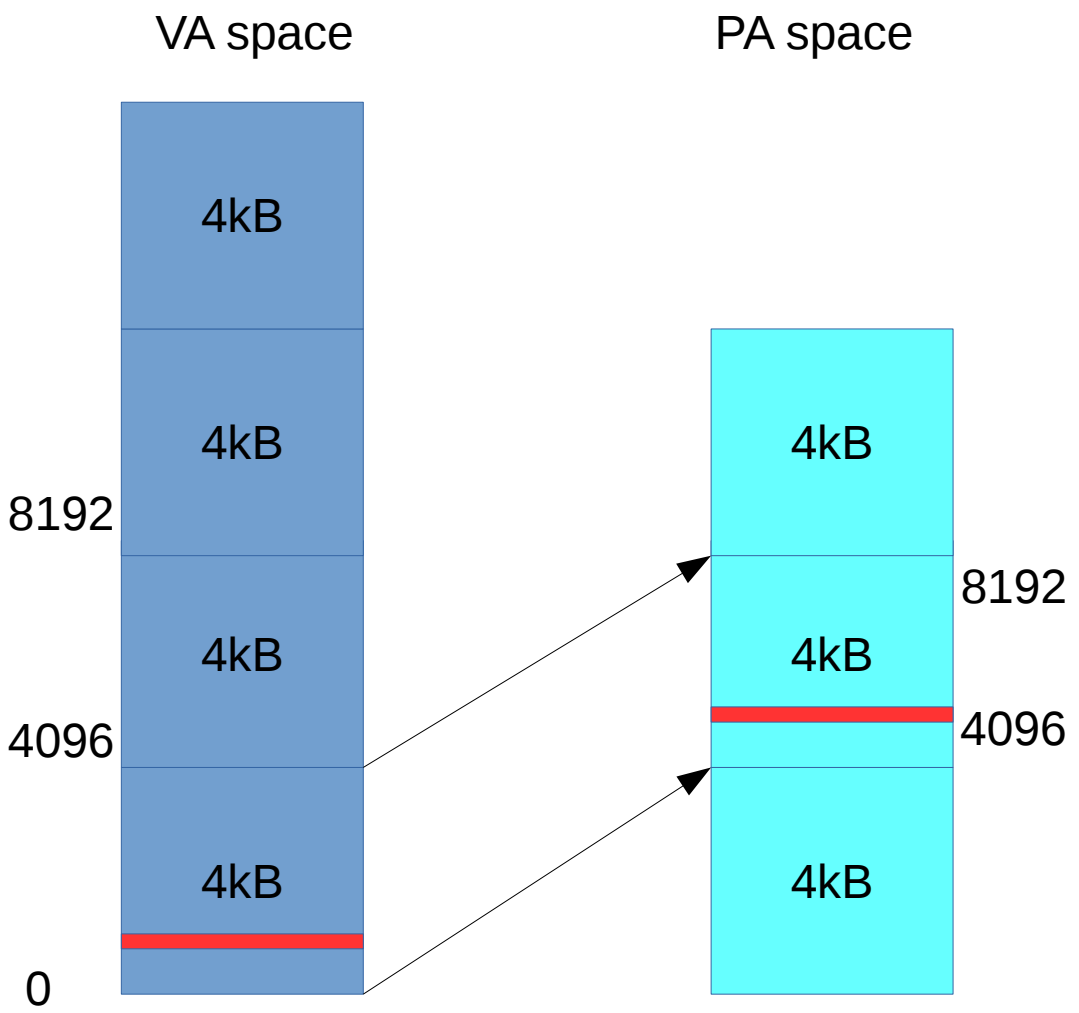
- 0x00003204



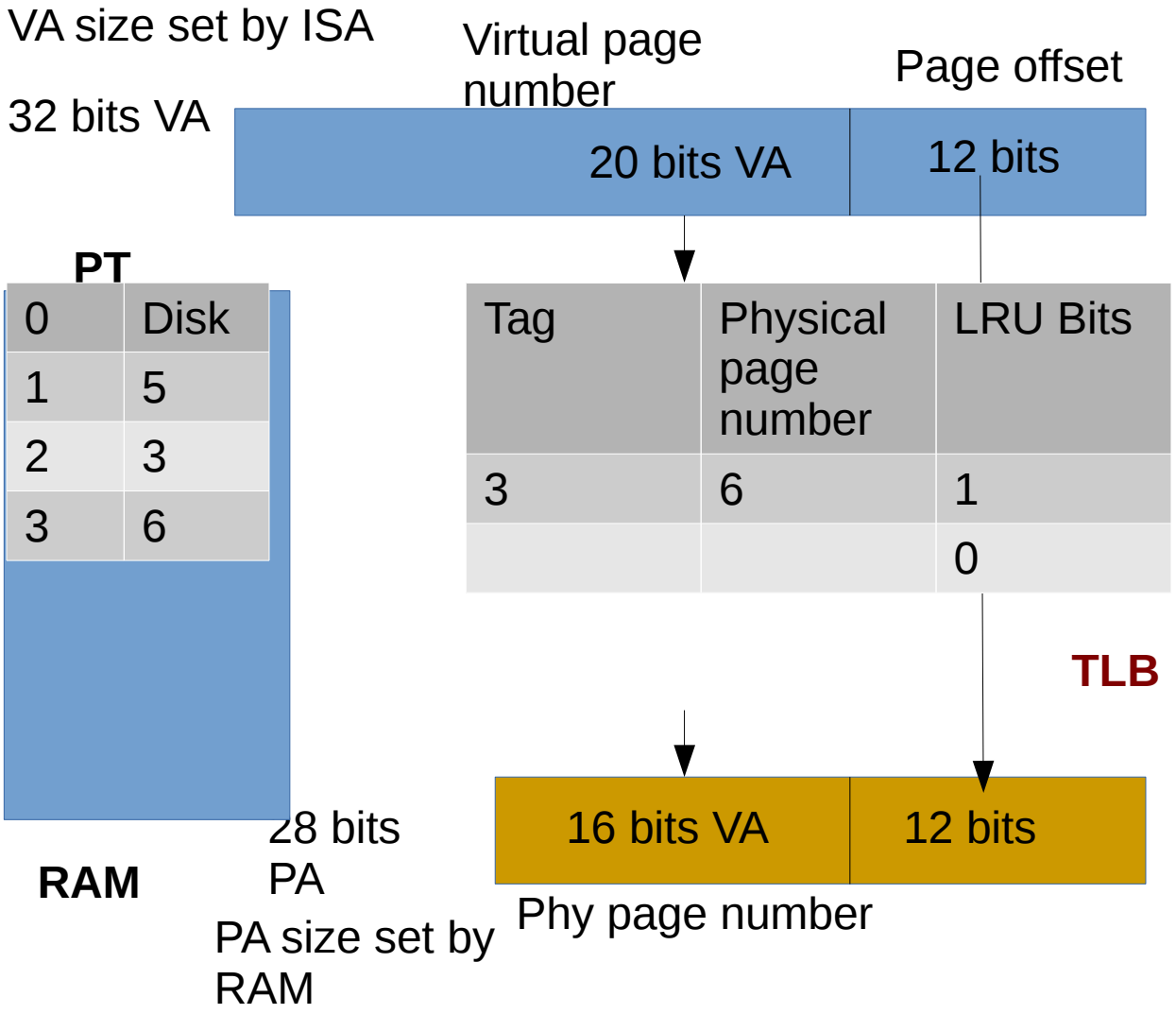

- **<u>Note that the translation is for a page, not for individual bytes or words</u>**
- 0x00003204 --> TLB Miss --> Exception --> OS handles the miss routine
- **Look up the VPN as an index into the PT in RAM**
- 3 --> 6
- OS: System instructions Update the TLB
  - Tag=3
  - Phy page number=6
- Next time --> TLB hit
- Next time if you get 0x00003206 --> Page number is 3. It will be a hit.

# What does the TLB store

- Note that the translation is for a page, not for individual bytes or words
- Say, Virtual address = 0: TLB Miss
- Go to the PT, fetch the translation, update the TLB
- Next, if we get a virtual address anywhere in that 4kB page (say, 2, 10, 4095), the translation will be a hit in TLB, as they share the same page number
- Remember TLB does not store data, it only stores translations
- Still need to go to cache or RAM to access data

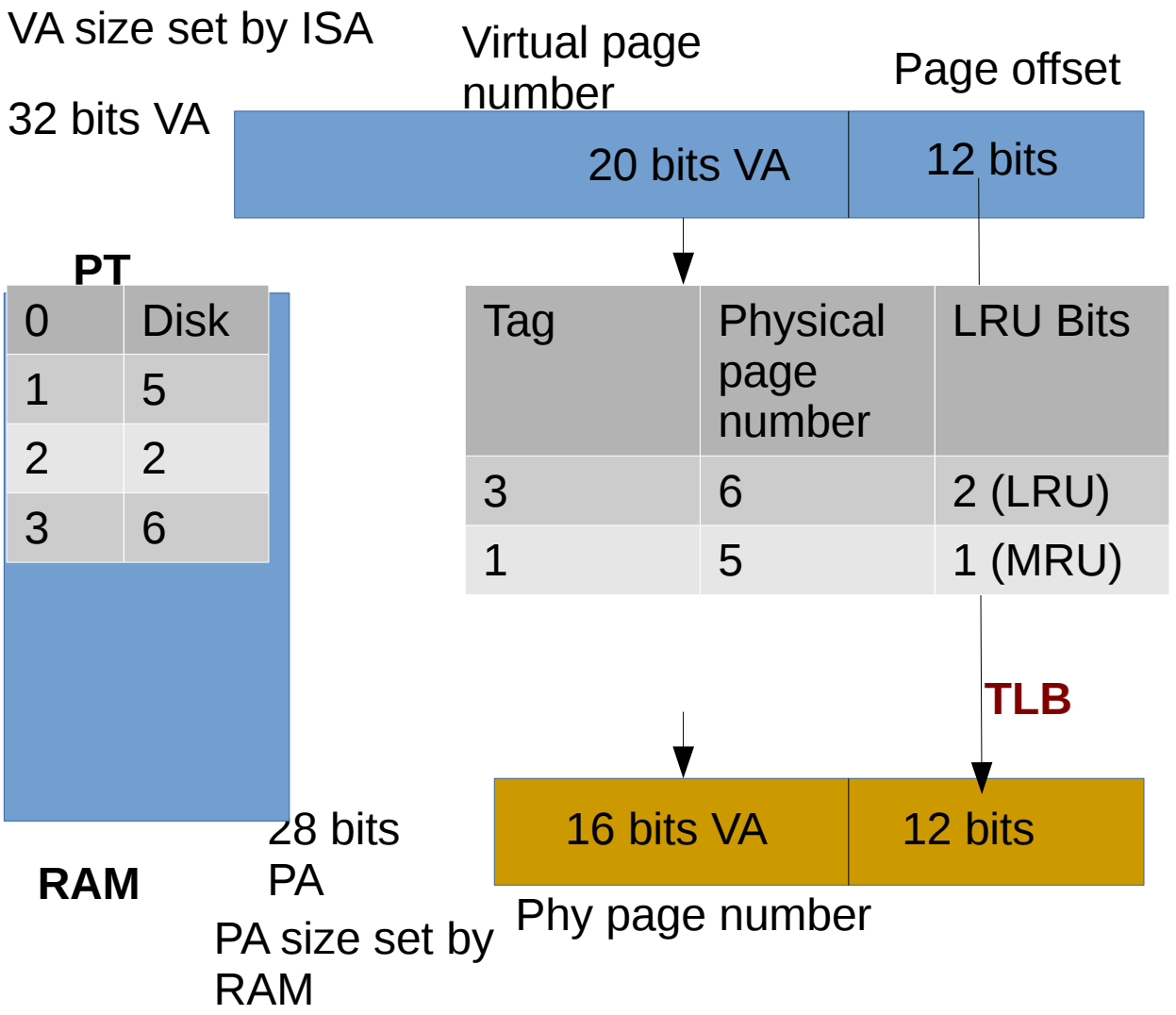


# Translation using TLB



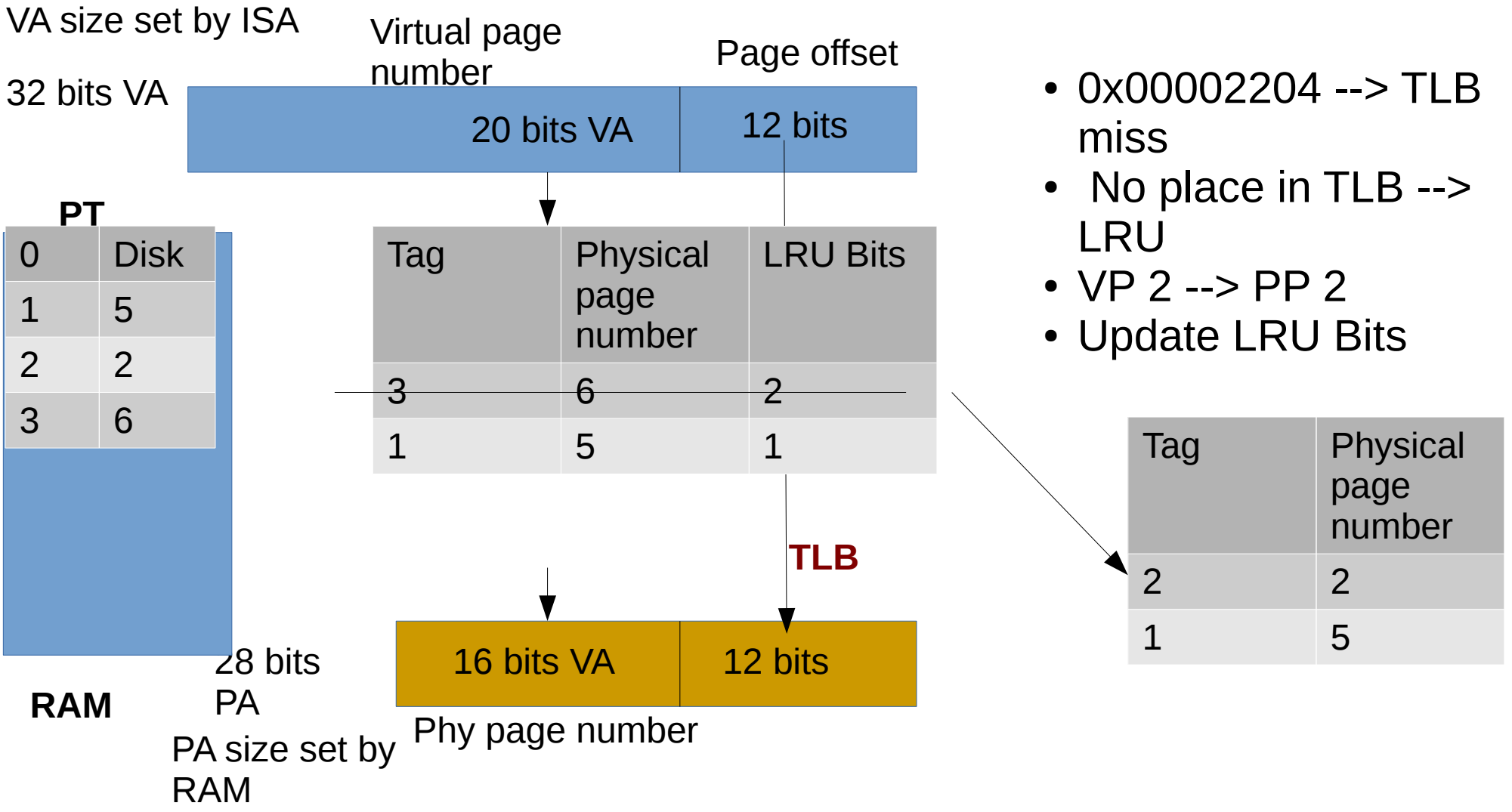

- 0x00001204

# TLB



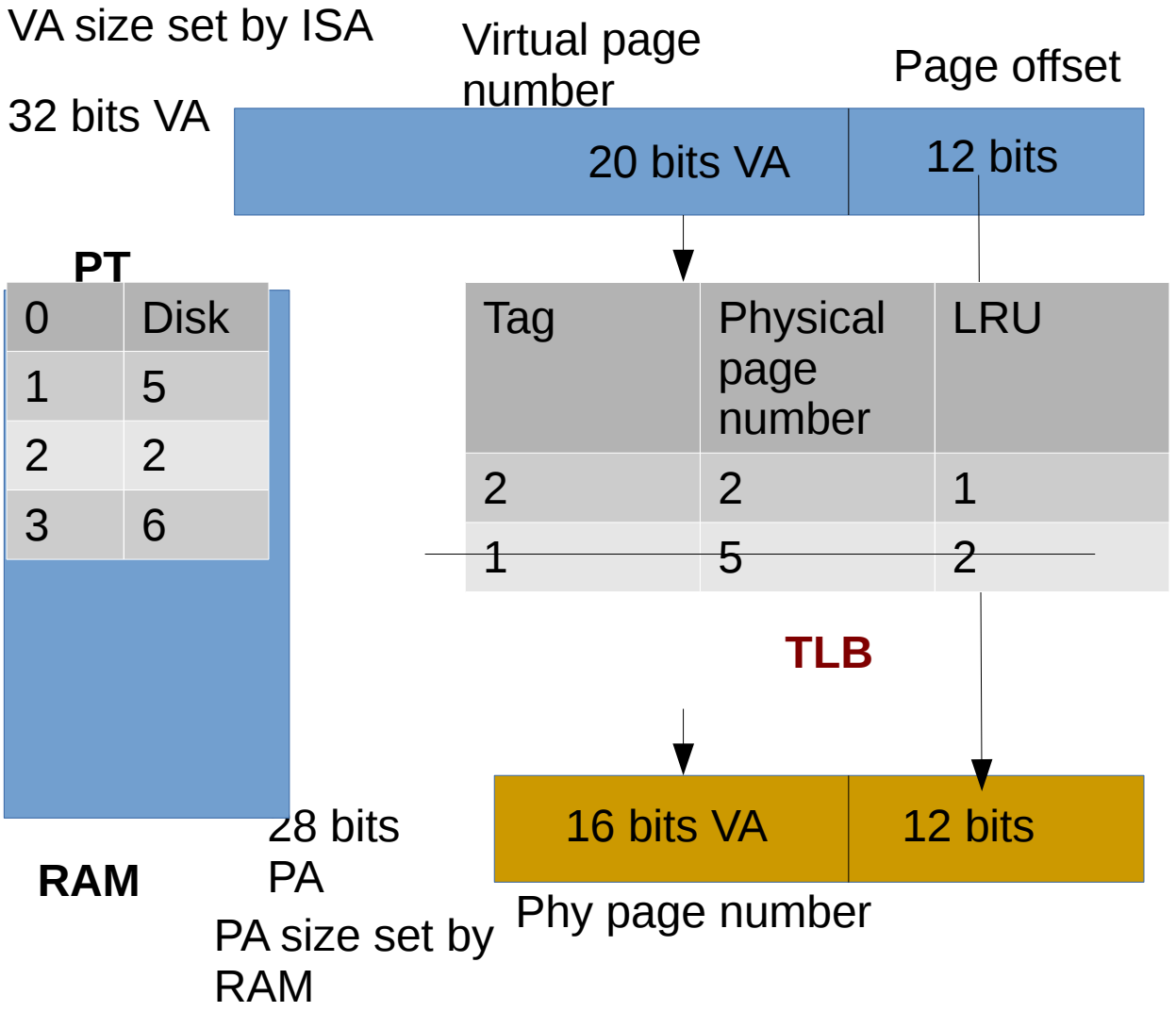

- TLB Miss
- 0x00001204 --> Update LRU Bits if needed

# TLB Miss



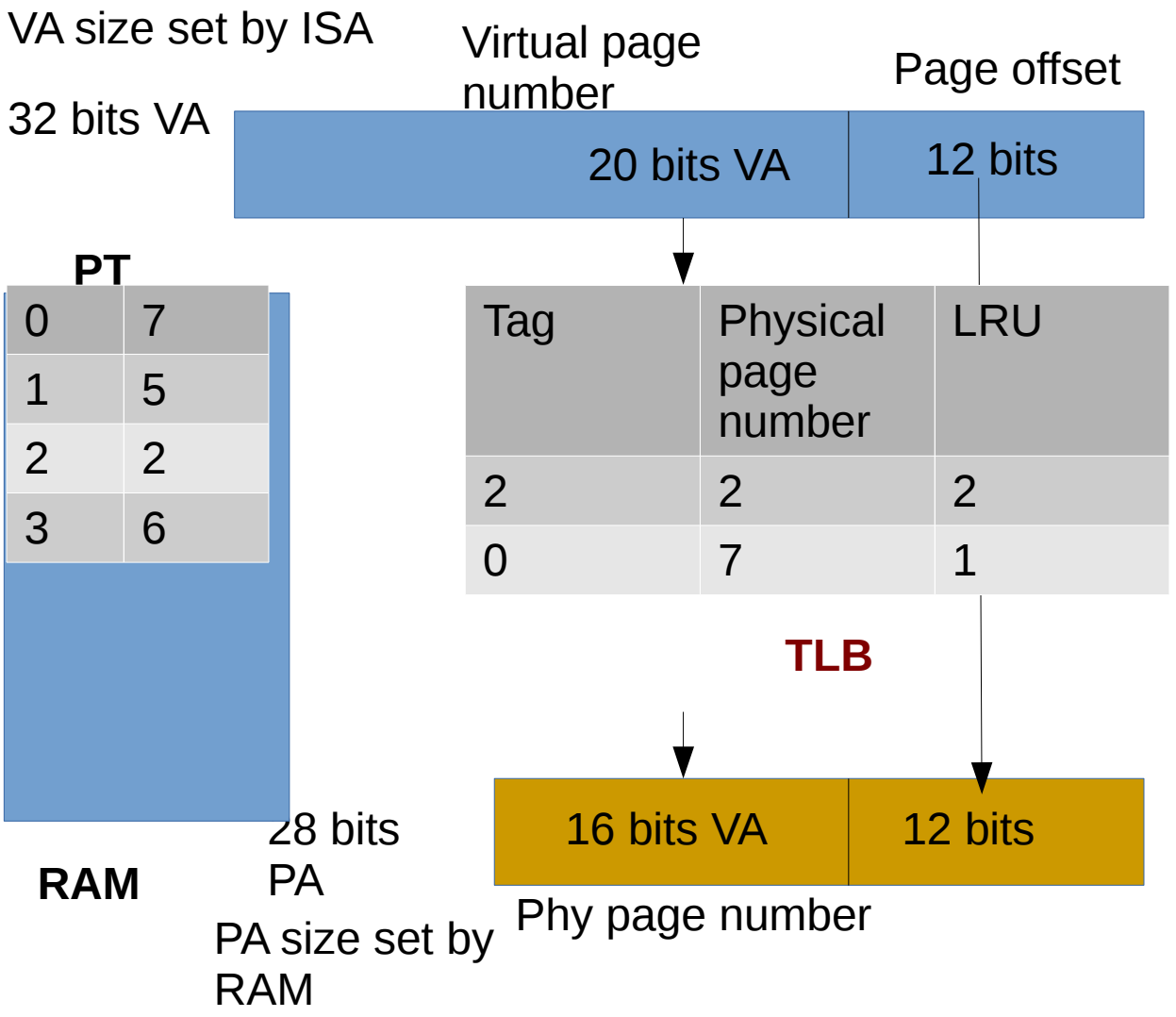

- 0x00002204 --> TLB miss
- No place in TLB --> LRU
- VP 2 --> PP 2
- Update LRU Bits

# TLB Miss




- 0x00000204 --> TLB miss
- No place in TLB --> LRU
- Evict Tag 1
- **Translation --> Page fault**
- Load to memory from Disk, assign a new Pg number- whichever is free update PT, update TLB

# TLB and PT status in the end




- No space in memory?
- Replace a page
- Check dirty bit of the page. Write back to disk before replacing
- Update page table, Update TLB entry

# Handling TLB misses and page faults

- Who handles TLB misses?
  - Miss exception -- OS – Software TLB Miss handling
    - Embedded processors
  - Processor? - Hardware TLB miss handling (similar to cache miss handling)- Faster
    - Eg- x86
- Page fault exception: OS

# Reduce TLB misses

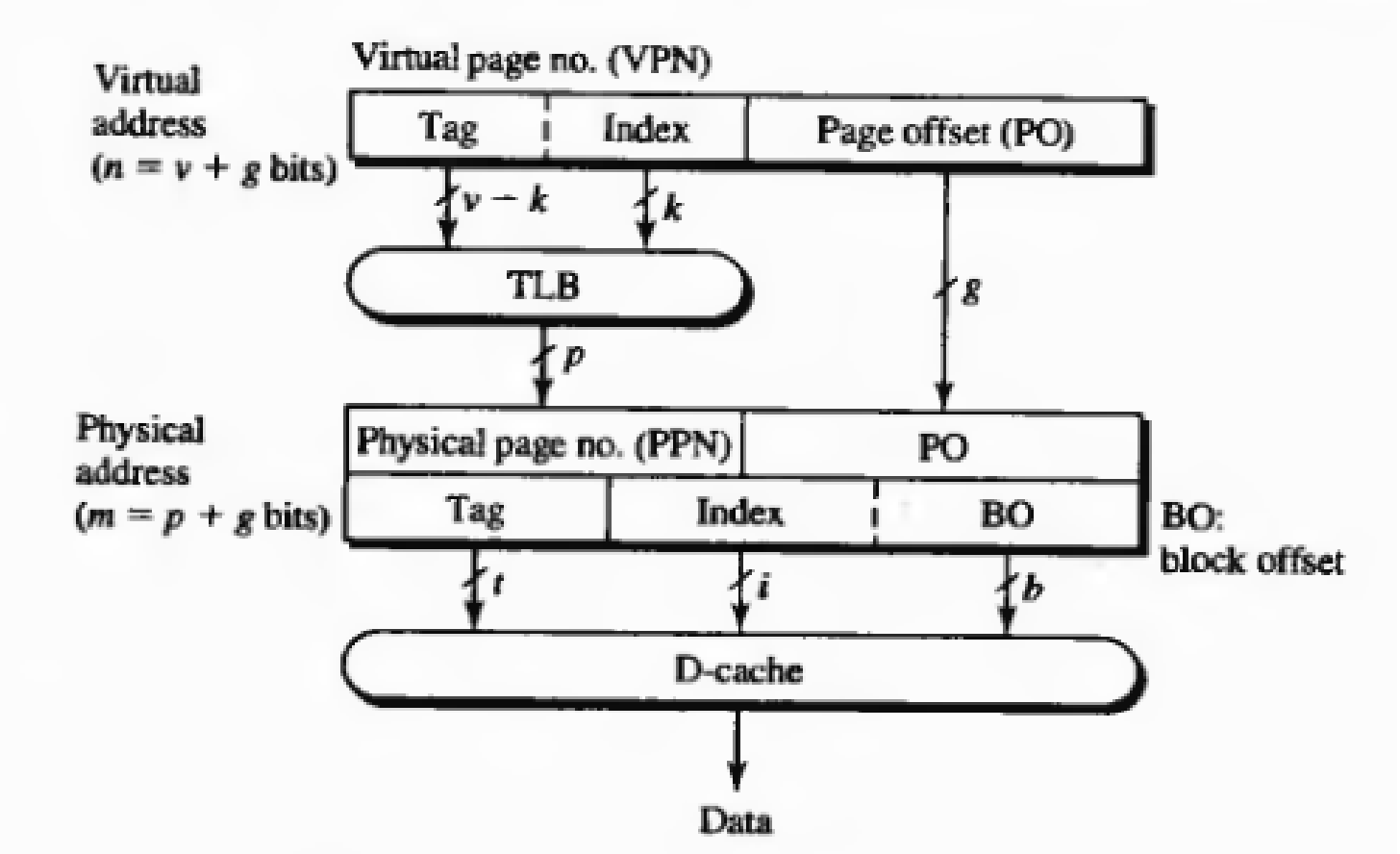
- Make pages larger
  - 64 4kB pages --> 256kB: Translate addresses to access 256kB of data
  - 32 2MB pages --> 64MB data can be covered
- Add a second TLB, like an L2 cache. Larger TLB
  - Eg: AMD Opteron

# Memory security

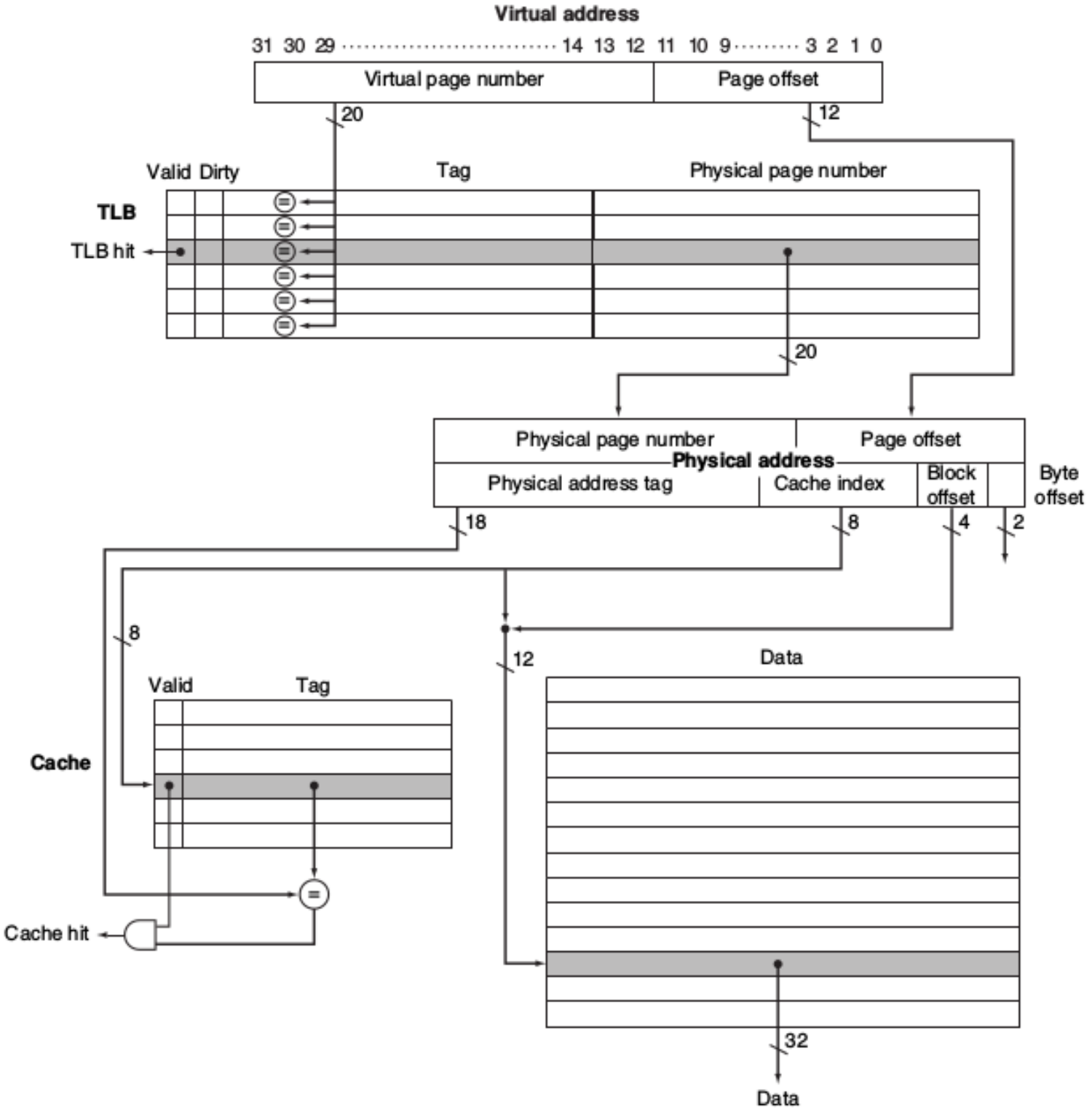
- Although multiple processes are sharing the same main memory, one process cannot write into the address space of another user process or into the operating system
- TLB: Write access bit: That provides this security

# TLB and Caches

# Physically indexed Physically tagged Cache



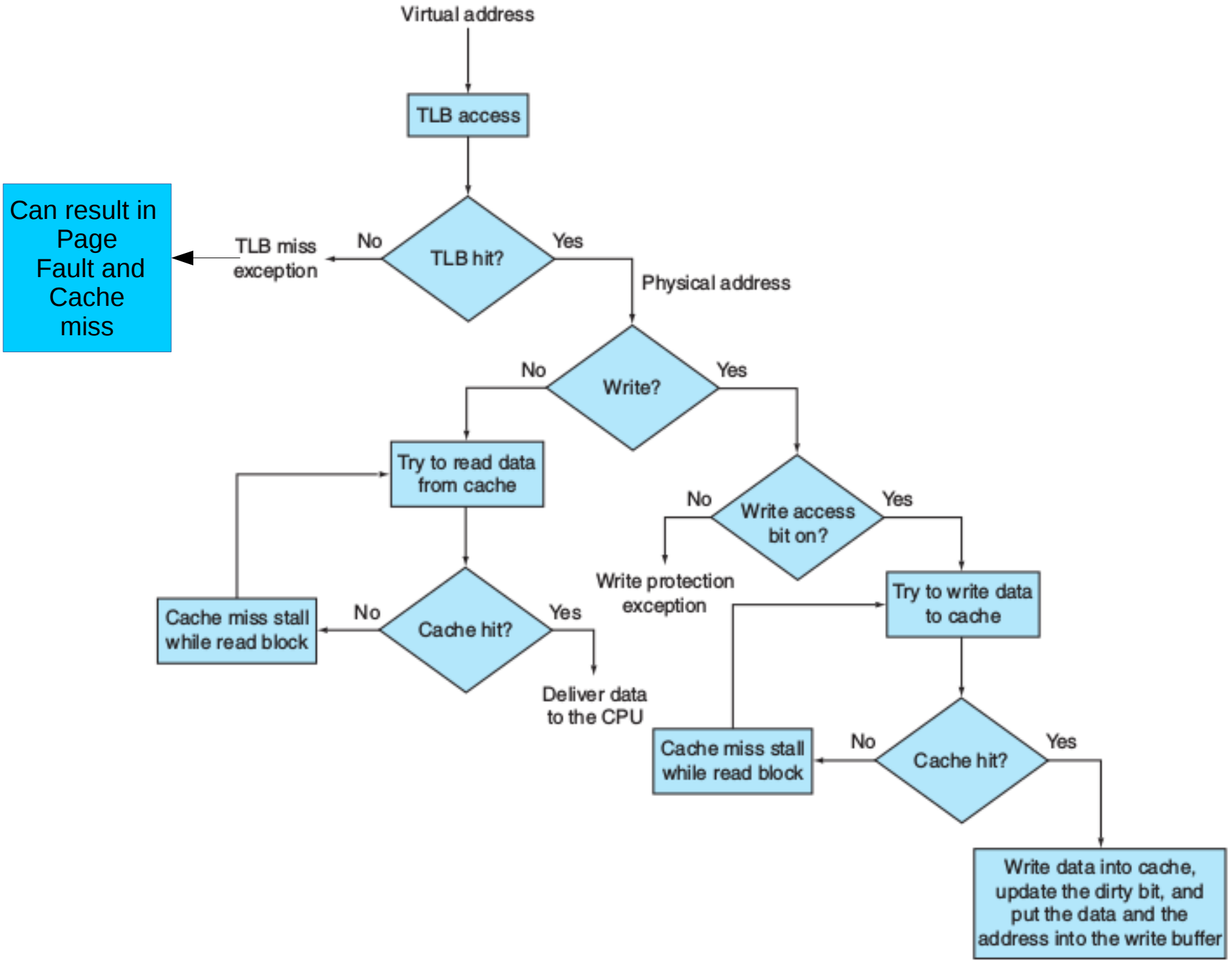
VPN = Tag.
What does it mean?




CO and design H&P

# Limitations of PIPT Cache

- TLB: Say 5 cycles: Has to happen before cache access
- Cache access: 5 cycles: Overall cache access: 10 cycles
- Serialises TLB and Cache


Virtual address
TLB access
TLB hit?
No
Yes
TLB miss exception
Can result in Page Fault and Cache miss
Physical address
Write?
No
Yes
Try to read data from cache
Cache hit?
No
Yes
Cache miss stall while read block
Deliver data to the CPU
Write access bit on?
No
Yes
Write protection exception
Try to write data to cache
Cache hit?
No
Yes
Cache miss stall while read block
Write data into cache, update the dirty bit, and put the data and the address into the write buffer

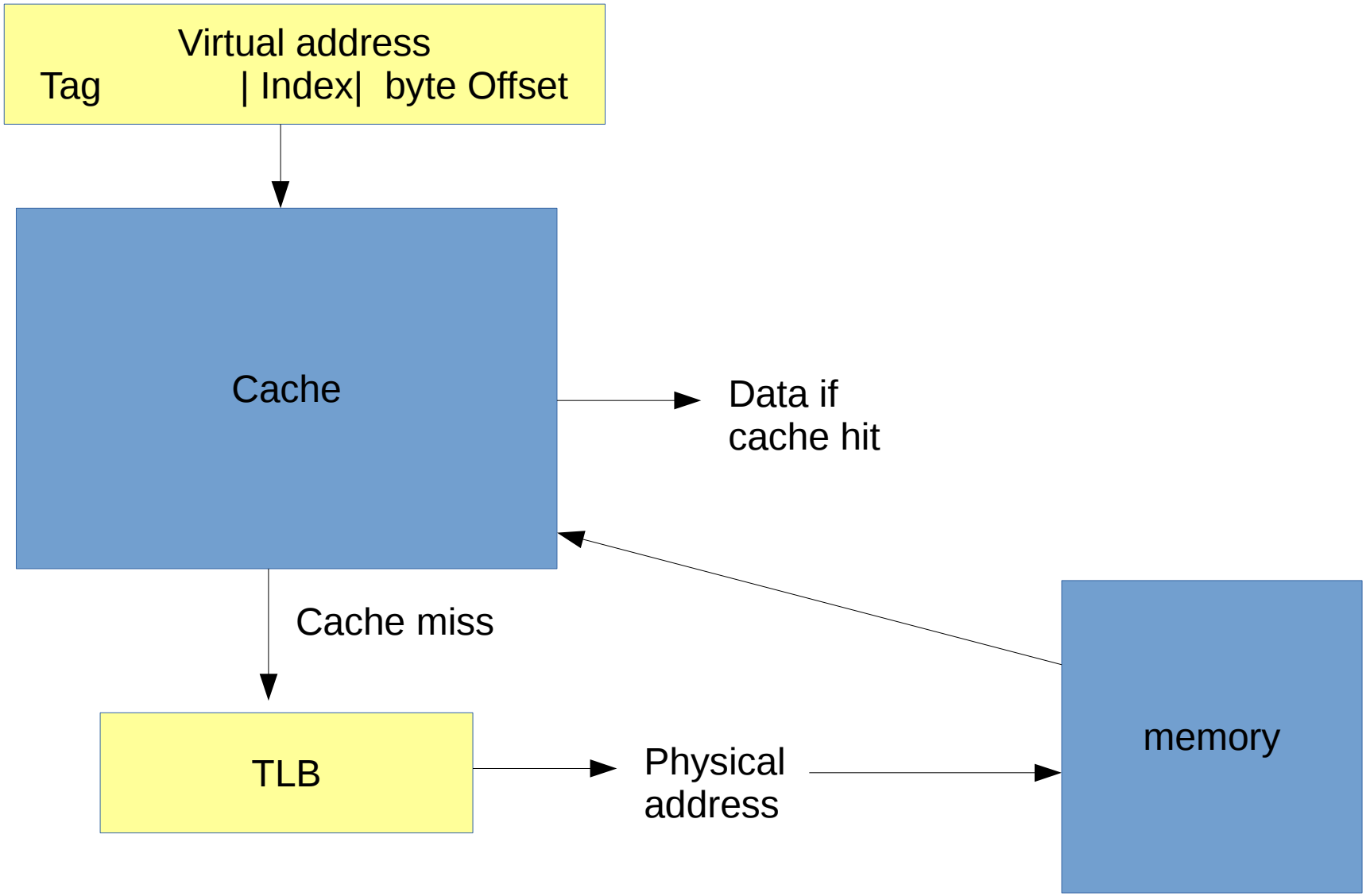
# Events in the TLB, virtual memory and PIPT cache

Translation present in TLB,
Page table is ‘dont care’

| TLB | Page table | Cache | Possible? If so, under what circumstance? |
|---|---|---|---|
| Hit | Hit | Miss | Possible, although the page table is never really checked if TLB hits. |
| Miss | Hit | Hit | TLB misses, but entry found in page table; after retry, data is found in cache. |
| Miss | Hit | Miss | TLB misses, but entry found in page table; after retry, data misses in cache. |
| Miss | Miss | Miss | TLB misses and is followed by a page fault; after retry, data must miss in cache. |
| Hit | Miss | Miss | Impossible: cannot have a translation in TLB if page is not present in memory. |
| Hit | Miss | Hit | Impossible: cannot have a translation in TLB if page is not present in memory. |
| Miss | Miss | Hit | Impossible: data cannot be allowed in cache if the page is not in memory. |
| Hit | x | Hit | Best case scenario, retrieve the data from cache |

Page not in memory, cannot be definitely present in the cache. Cache must miss

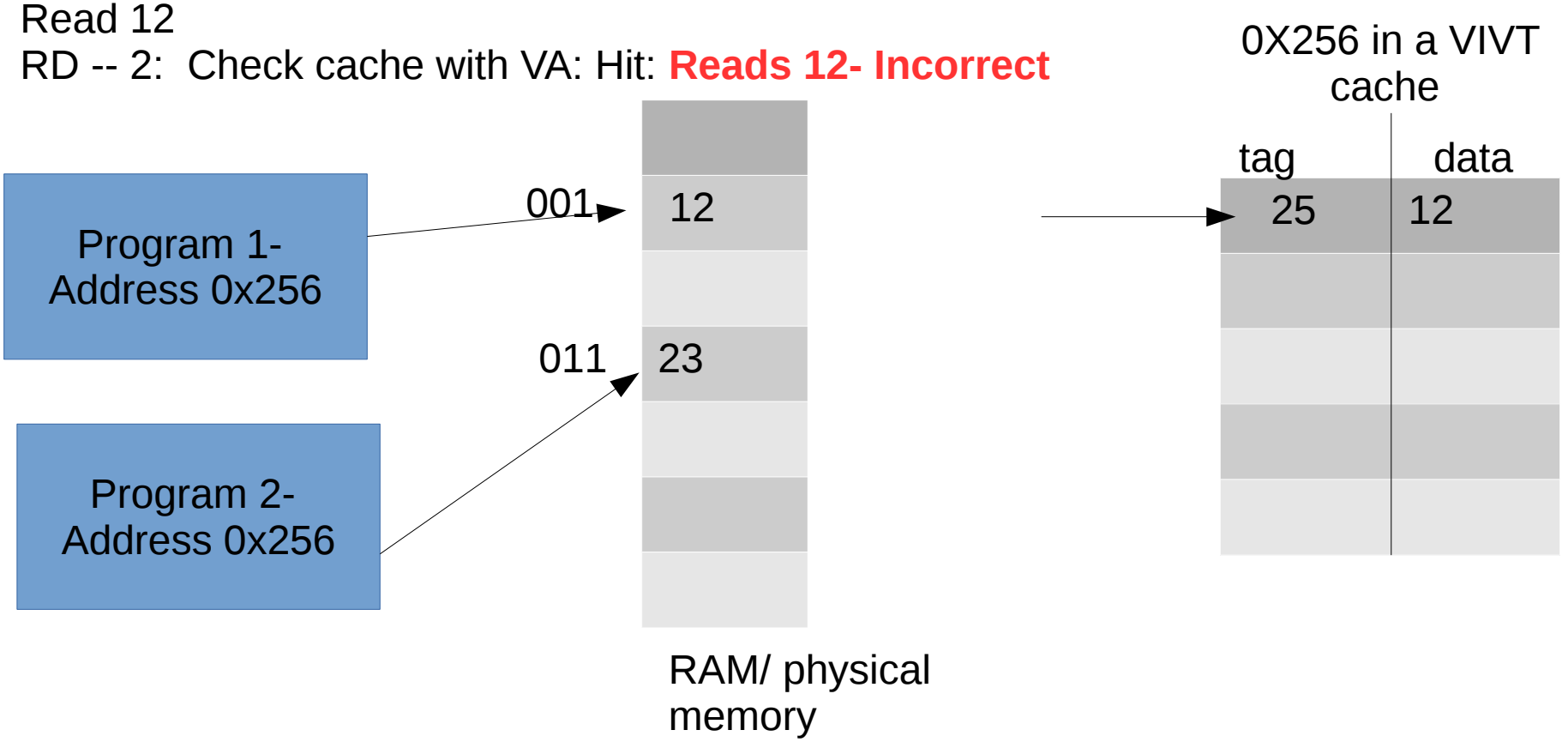
# Virtually indexed Virtually tagged Cache



# VIVT Cache

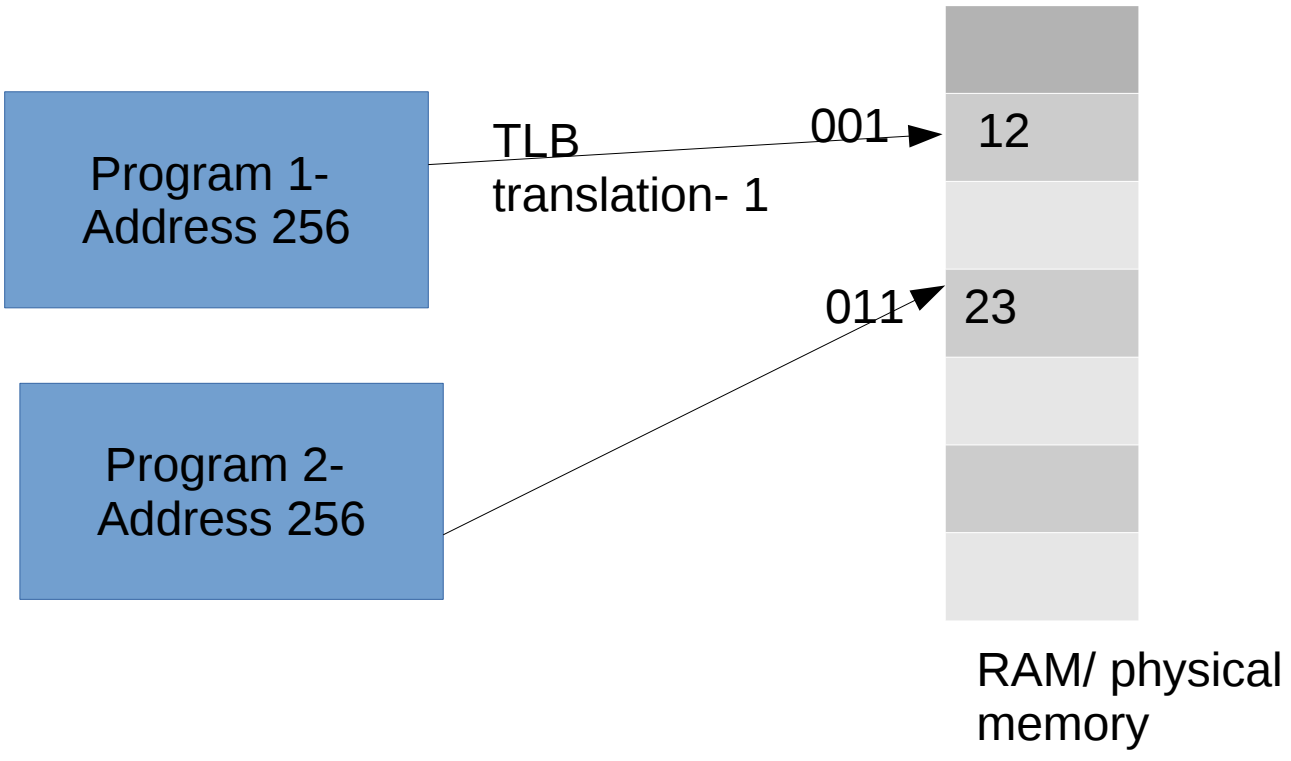
- No TLB latency to access cache. TLB not part of the critcal path
- Limitation:
  - 2 processes can have the same Virtual address, but map to different physical addresses. TLB translations will be different

RD -- 1 : First check cache: Cache miss: TLB: 256 --> 001: Read 12
RD -- 2: Check cache with VA: Hit: **Reads 12- Incorrect**



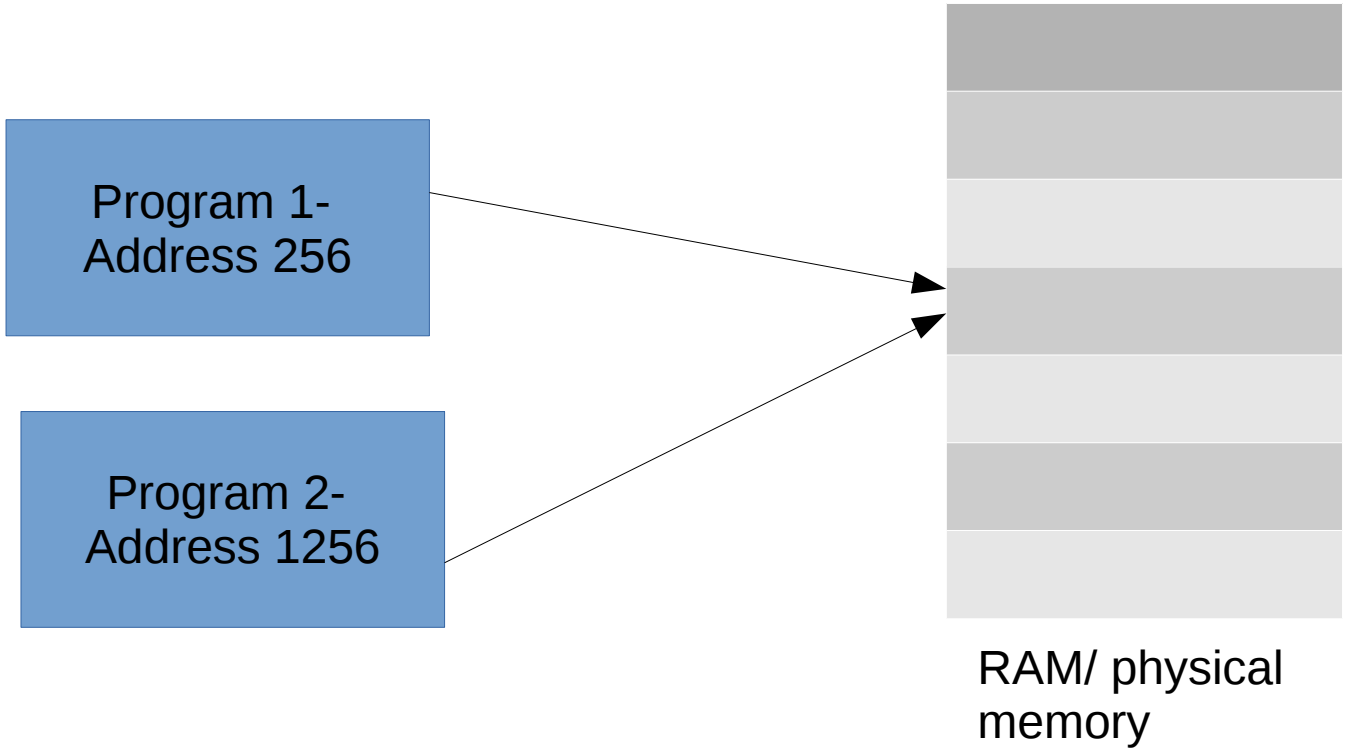
# VIVT Cache

- When a process/context switch happens, **the entire VIVT cache and TLB will have to be flushed** so that correct data is loaded
  - Can be solved using a **process identifier** along with TLB/Cache entry



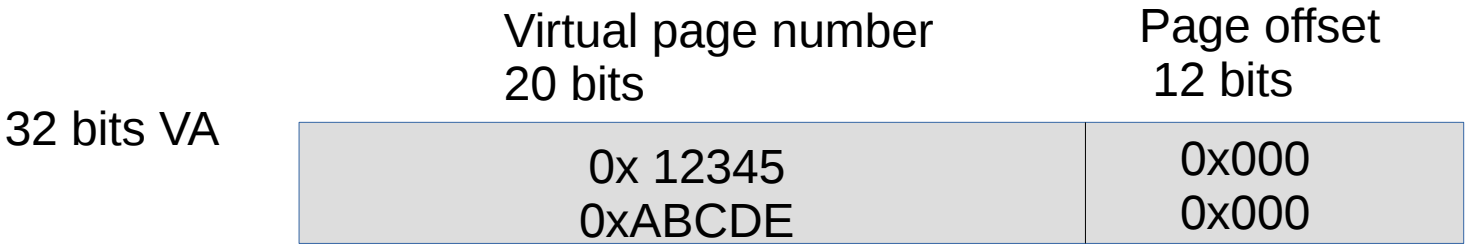
# Aliasing

- Different VA, but same PA – sharing memory

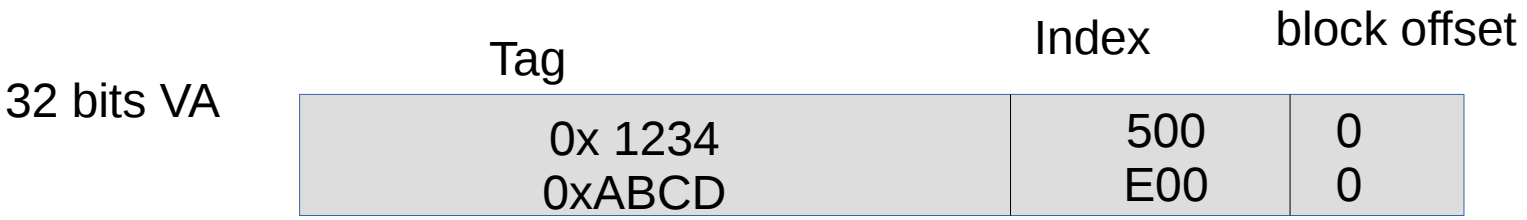


# Aliasing

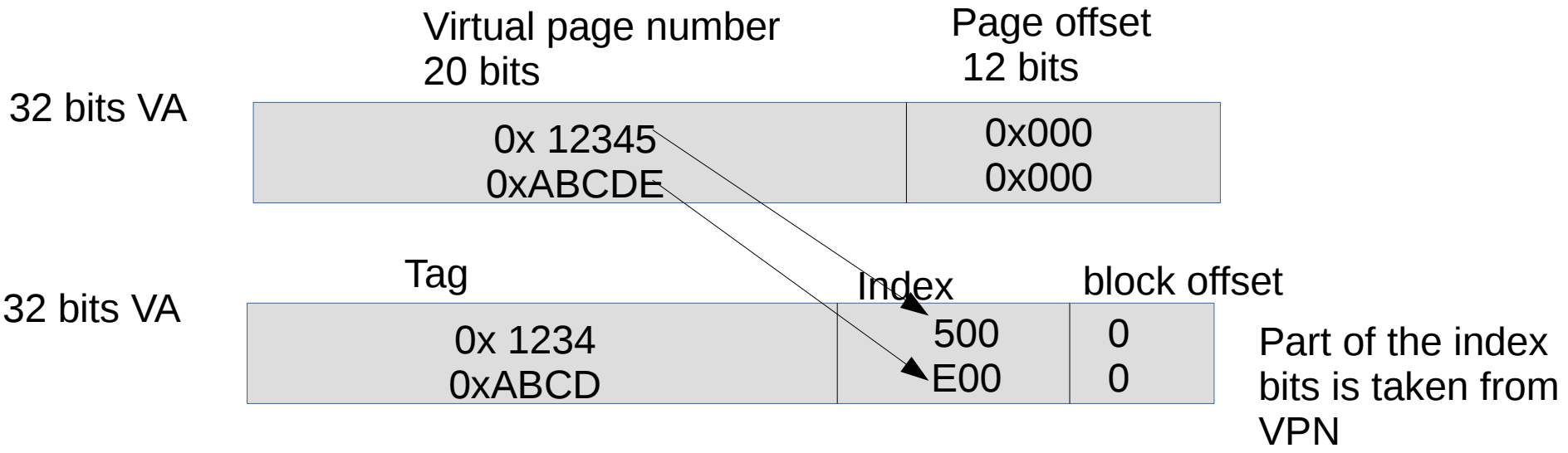
Process 1: Virtual Address A: 0x12345000
Process 2: Virtual Address B: 0xABCDE000
4kB page: 12 bit page offset




Direct mapped cache: 16 byte block: 4 bits of byte offset
4096 cache lines: 12 bits of index
Index: Virtual index
Tag: Virtual tag

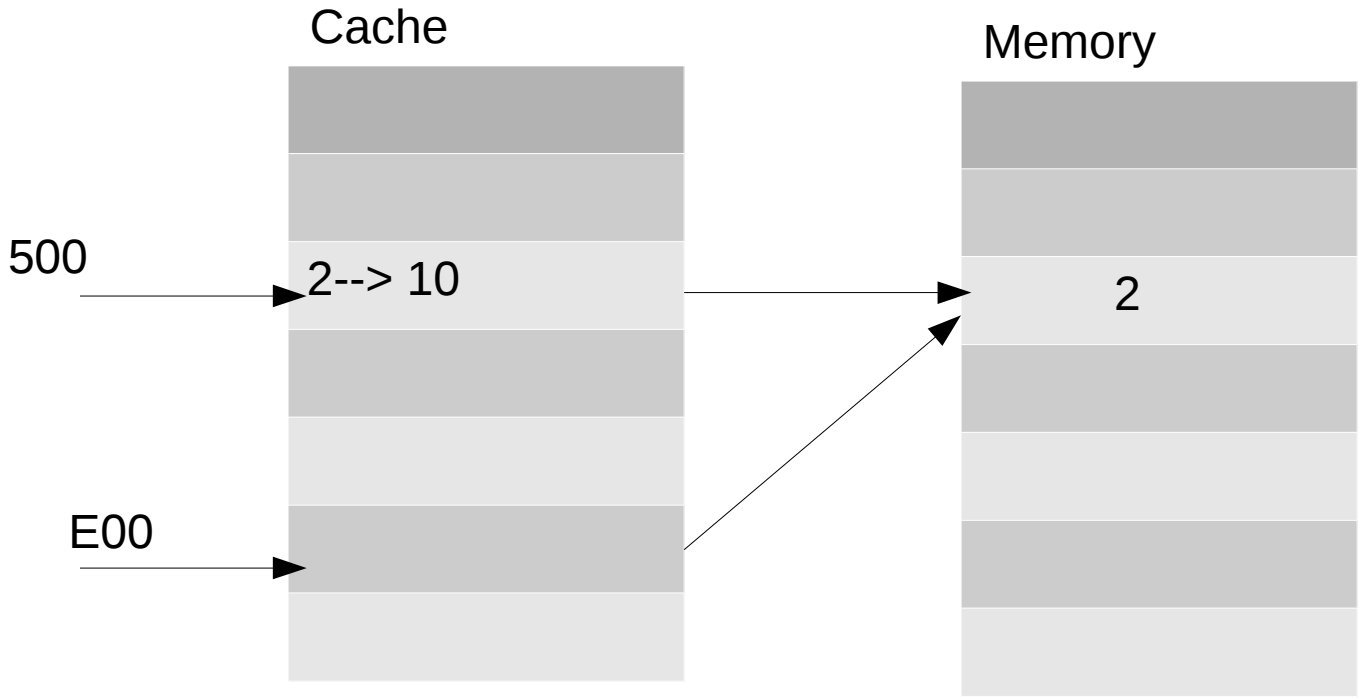


# Aliasing

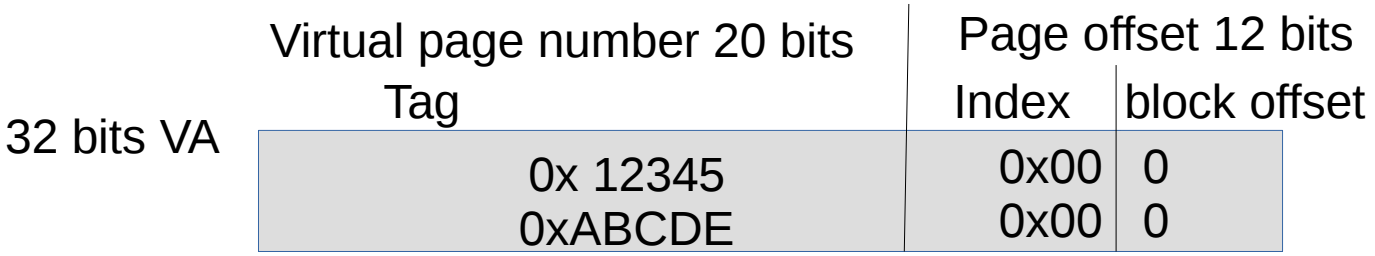



RD- A –index into 500:
Miss- Read 2
WR- A – Hit: Write 10. Dirty bit=1
RD-B –Index into E00:
Miss: **Read 2 – Inconsistent data**

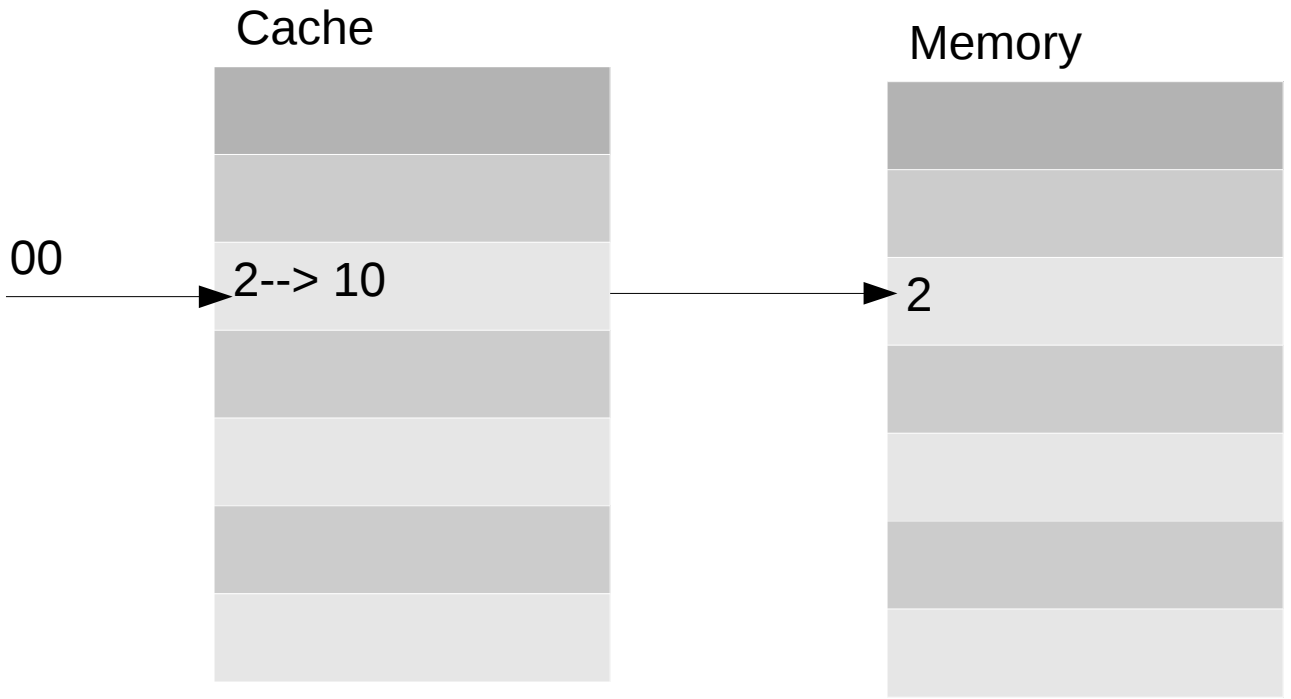
**Problem: Different indices inspite of mapping to same physical memory**



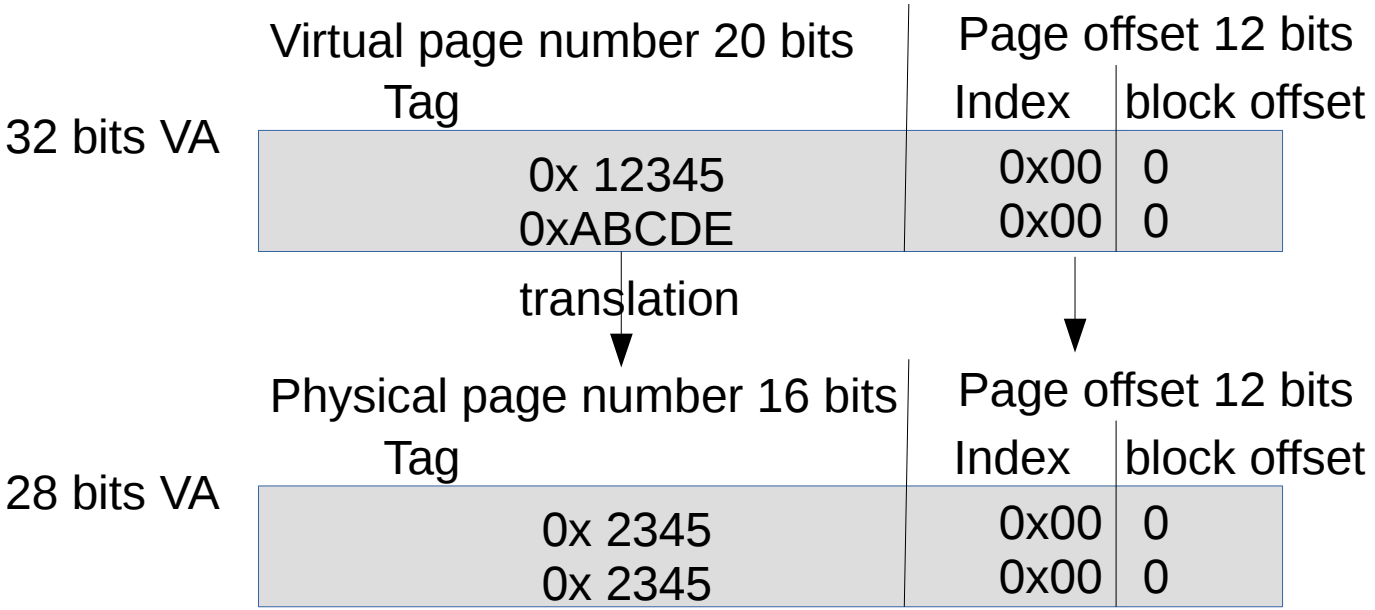
# Aliasing solution




Problem of Mapping to different indices in the cache:
- Need to have same index
- Note that **the 2 virtual addresses mapping to the same physical address will have the same page offset --> Pick the index bits from page offset**
**- What about tag?**



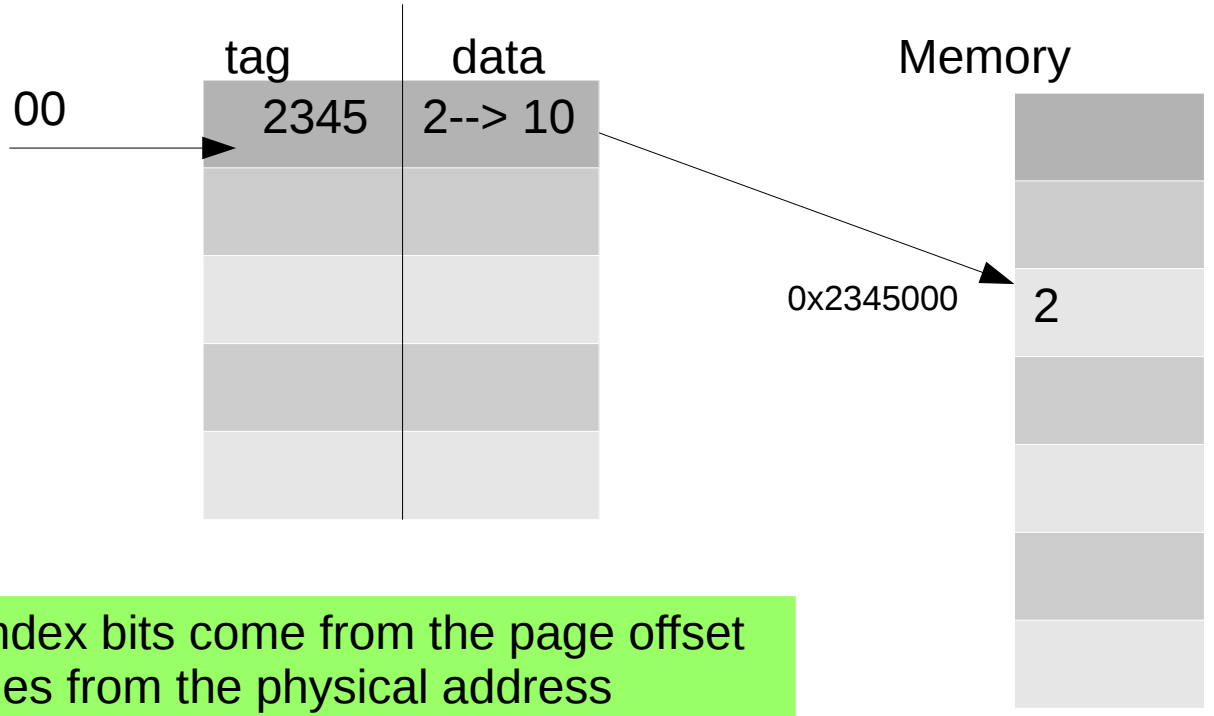
# Aliasing solution




RD- A –index into 00: Miss- Read 2
WR- A – Hit: Write 10. Dirty bit=1
RD-B –Index into 00: Miss: **Read 10**

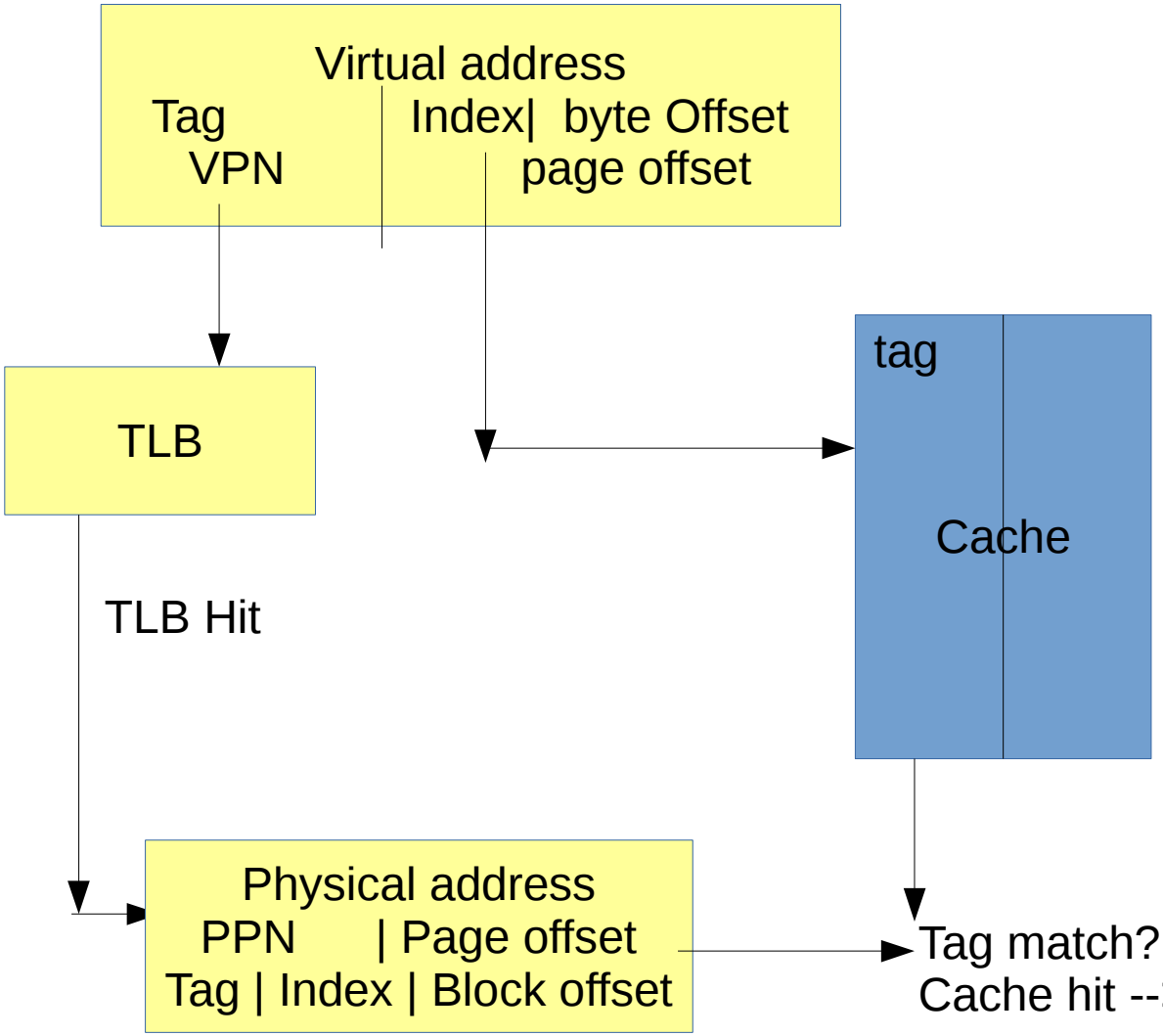
**Tag: Physical tag: Obtained after address translation. Both Virtual addresses will have the same tag**

**Also note that the index bits in virtual and physical addresses are the same**




No aliasing if the index bits come from the page offset and tag comes from the physical address

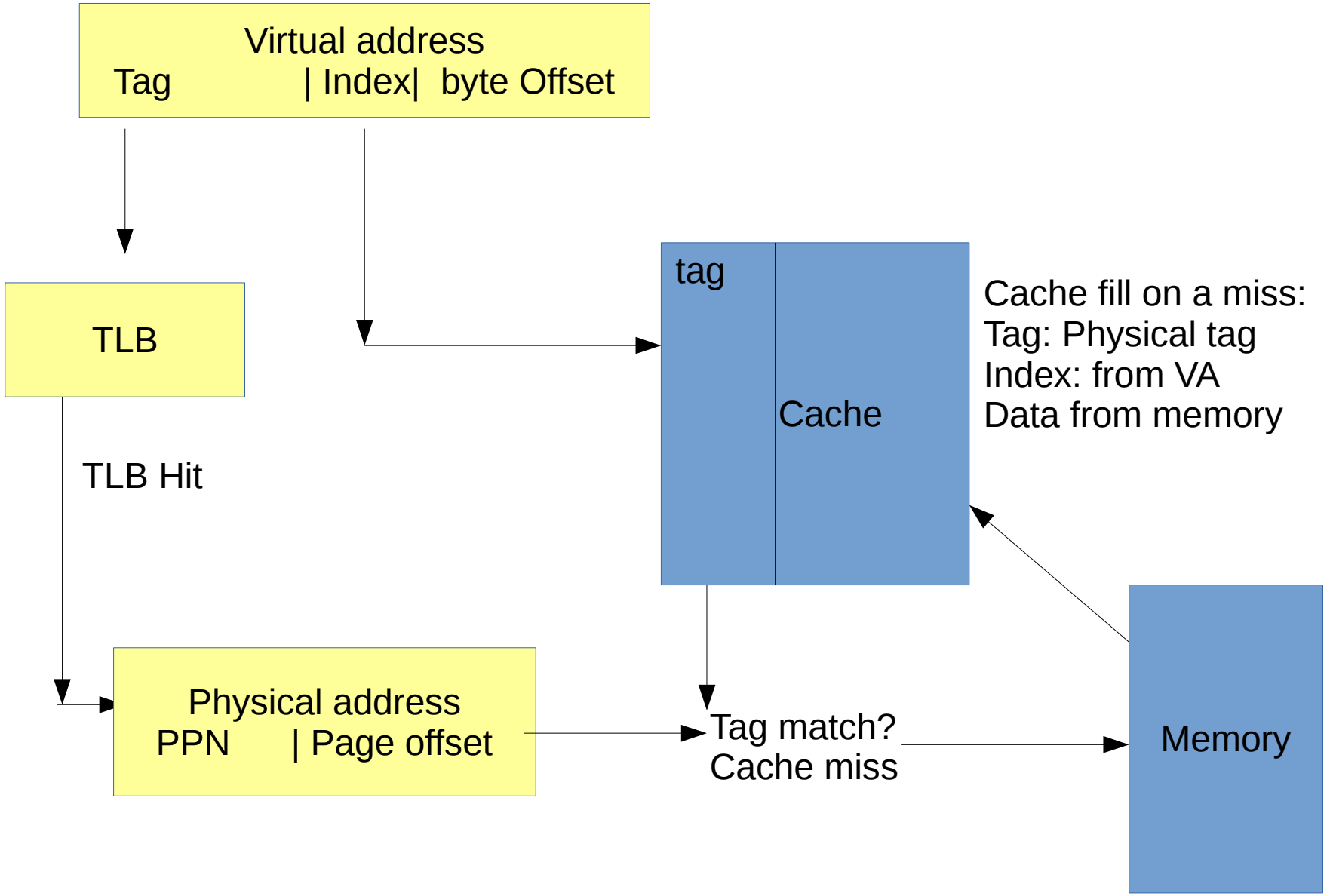
# Virtually indexed Physically tagged Cache



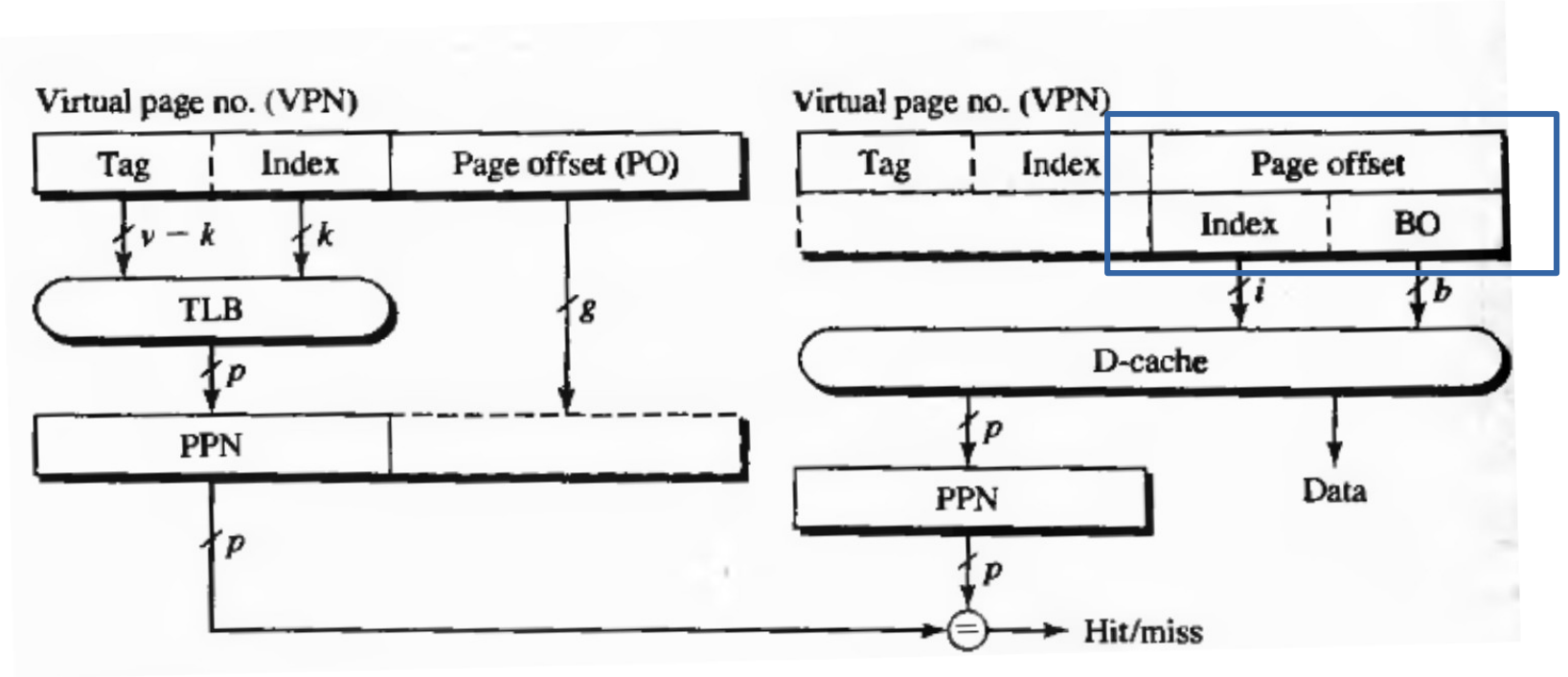

**The index bits in virtual and physical addresses are the same**

Why are we indexing using the virtual address, and not physical address? Obtaining Physical address will need translation (TLB/Page table). Avoid this delay

# Virtually indexed Physically tagged Cache



# VIPT Cache

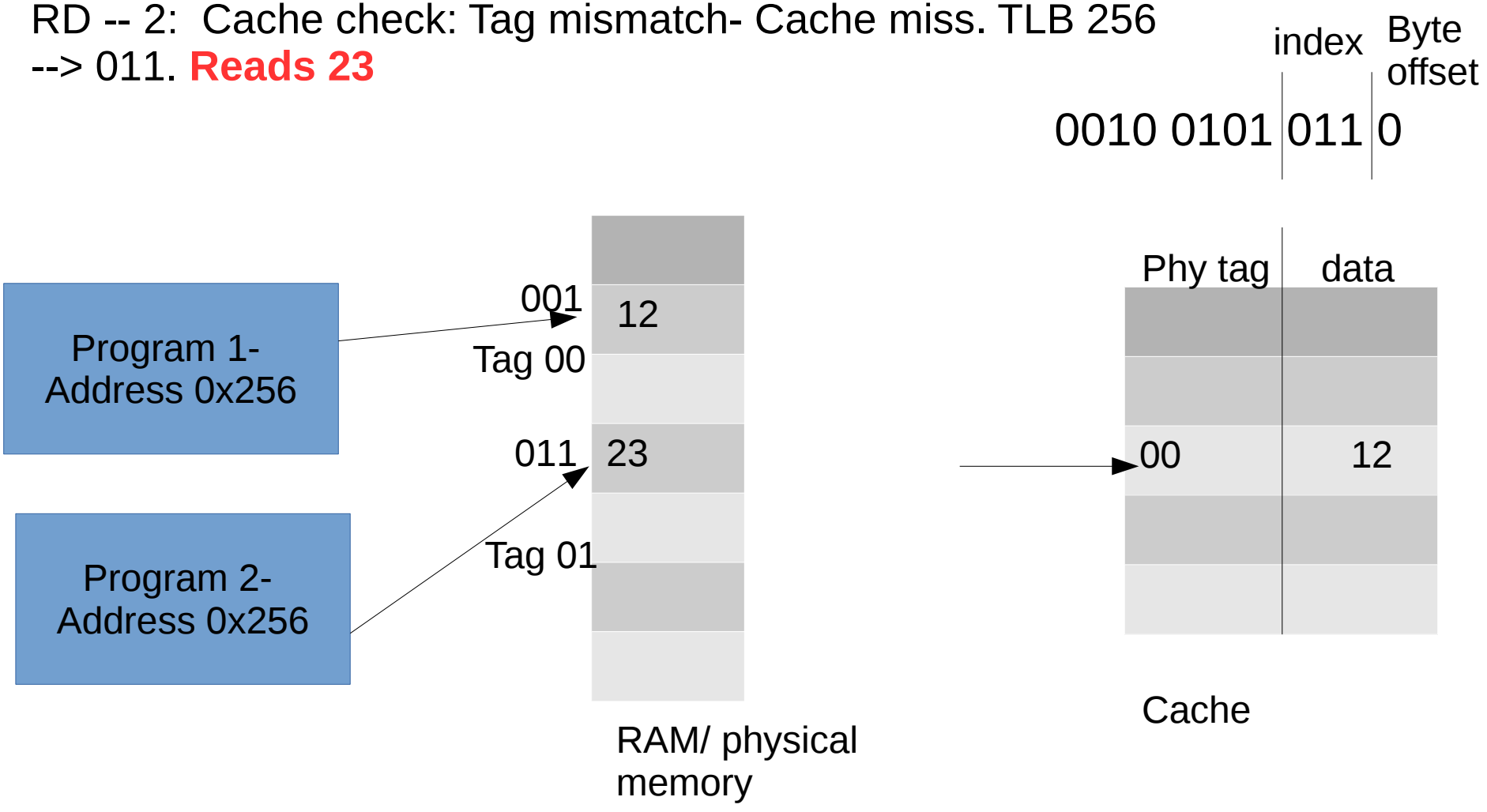



- Index the cache using the virtual address. Index bits come from the Page offset- **This will limit the cache size**
- Tag will contain the physical address's tag obtained after address translation
- Cache hit check: Compare tags in the cache with the tag obtained from TLB
- Advantage: TLB and Cache look-ups are parallel.
  - No need of cache flush on context switch – ??

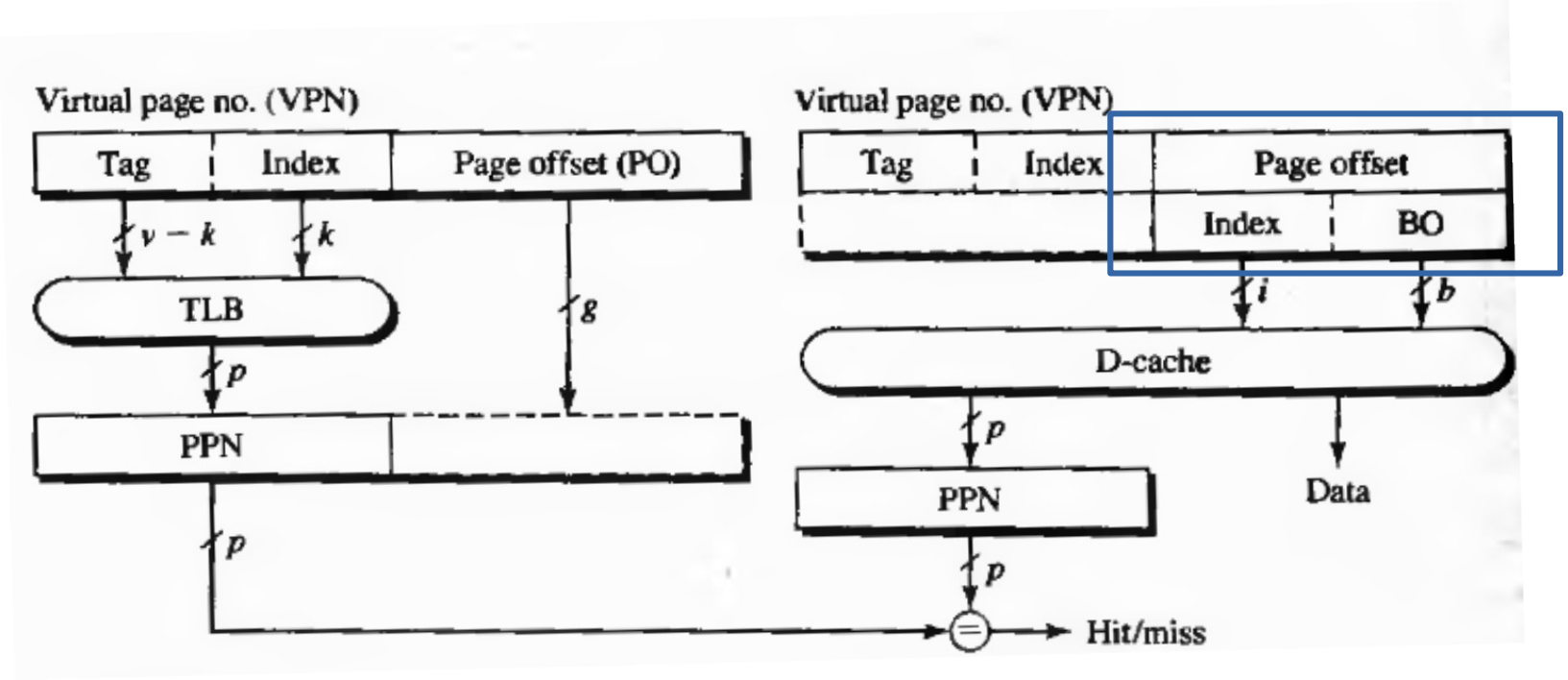
# VIPT Cache

- When a process/context switch happens, **the entire VIVT cache and TLB will have to be flushed** so that correct data is loaded

RD -- 1 : Cache miss: TLB: 256 --> 001: Tag: 00: Read 12
RD -- 2: Cache check: Tag mismatch- Cache miss. TLB 256 --> 011. **Reads 23**



# VIPT Cache size




- 4kB page: Needs 12 bits of page offset
- A 16Byte block size in the cache needs 4 bits of byte offset
  - If index has to come from the page offset to avoid aliasing, index bits = (Page offset – Byte offset) --> Max: 8 bits index
  - 2^8 lines or sets in the cache = 256 sets – maximum size

# Eg

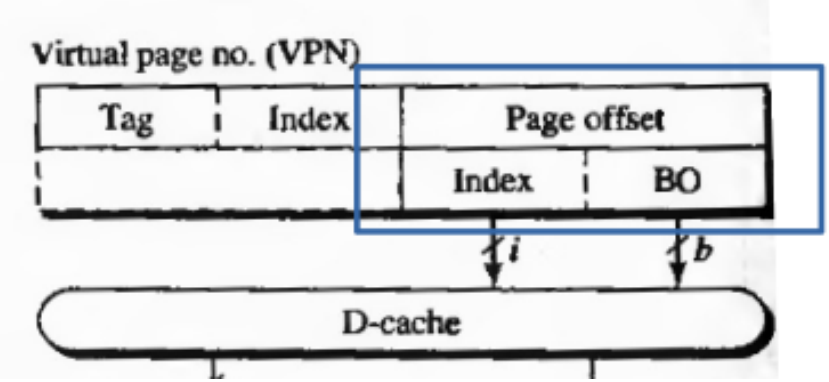
- VIPT cache: 4 way Set associative, 16 Byte block size, 4kB page. What is the maximum cache size to avoid aliasing

# Eg

- VIPT: 4 way Set associative, 16 Byte block size, 4kB page. What is the maximum cache size to avoid aliasing
  - 4kB page: 12 bits of page offset
  - 16 B block size: 4 bits byte offset
  - Index: 8 bits
  - 2^8 sets * 2^4 bytes per block * 2^2 ways
  - 2^8 sets * 2^2 blocks per set * 2^4 bytes per block
  - 2^14 bytes: Max size: 16kB

# ..cont

- VIPT: 4 way Set associative, 16 Byte block size, 4kB page. What is the maximum cache size to avoid aliasing
  - 4kB page: 12 bits of page offset
  - 16 B block size: 4 bits byte offset
  - Index: 8 bits
  - 2^8 sets * 2^4 bytes per block * 4 way
  - 2^(8+4) * 4 ways
  - 2^(index bits + byte offset bits) * 4 ways
  - 2^ (page offset) * No of ways
  - Page size * No of ways = Max cache size




- Eg: Pentium 4: 4 way SA cache, 4kB page--> 16kB L1 size
- Haswell: 8 way SA, 4kB page --> 32kB L1

# Reading task

- Computer architecture- quantitative approach: H&P (4th edition): 5.6 Putting It All Together: AMD Opteron Memory Hierarchy
- Solve the homework practice problems