

Lecture 18:

Memory – Virtual Memory (Implementation)

Hunjun Lee

hunjunlee@hanyang.ac.kr

Is Reducing the Table Size Sufficient? → Do More!!!

- ◆ Problem: even with optimizations, we still need a large enough page table which easily exceeds MB-scale
 - Every possible memory reference requires translation
 - MB-scale table is large considering the L1 cache size

64KB

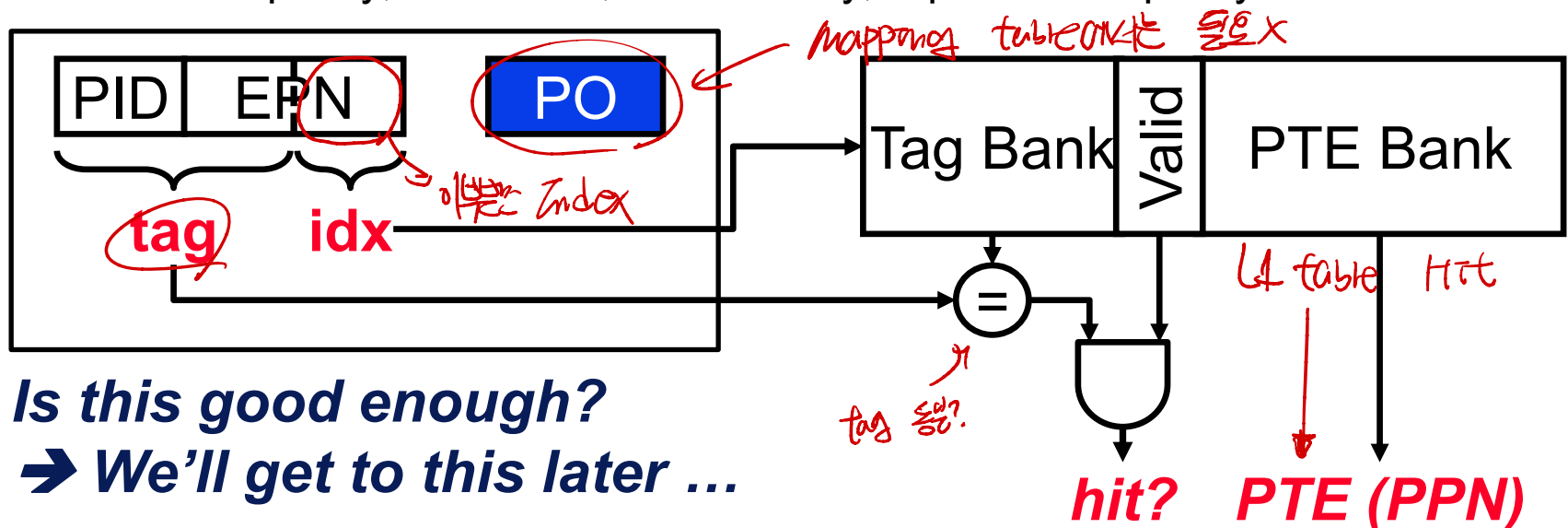
- ◆ Solution: How to resolve this? You know the solution already!!!

Mapping table 2/3/3 Cache 2/3/3

Translation Look-Aside Buffer (TLB)

◆ TLB: cache of recently used translations

- Same type of “tagged” lookup structure as caches and BTBs
- Given a VPN, returns a PTE (i.e., PPN & protections)
- TLB entry
 - Tag: address tag (from VA), PID
 - PTE: PPN, protection bits + Misc: valid, dirty, etc.
- Similar design considerations as caches
 - capacity, block size, associativity, replacement policy

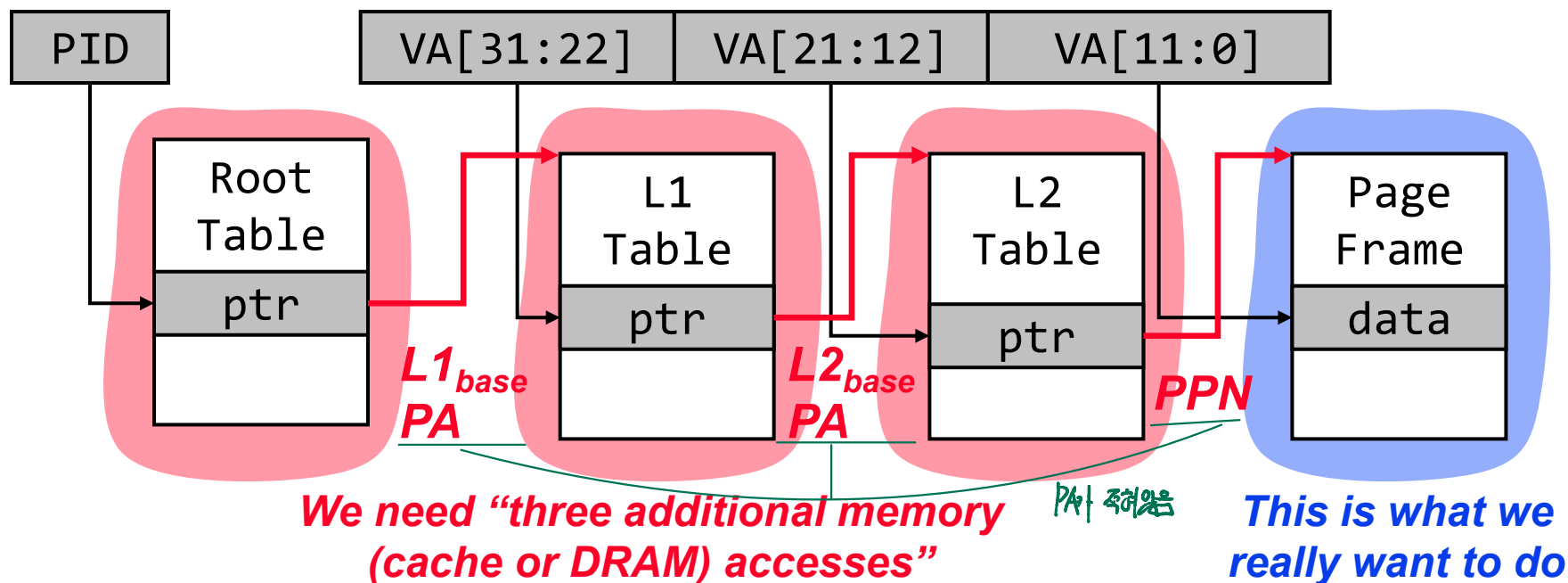


TLB Design Choices

- ◆ Separate I-TLB & D-TLB + there are multi-level TLBs (as in cache!)
- ◆ Capacity (C): L1 I-Cache vs. L1 I-TLB capacity?
 - Should cover the data in the L1 I-Cache (e.g., 64 KB)
 - A minimum size $(64 \text{ KB} / \text{page size}) * \text{safety factor} (2 \sim 8) \rightarrow 32 \sim 128$
- ◆ Block Size (B): Is there a spatial locality between page accesses? \rightarrow Not likely ...
 - Typically, a single PTE per entry
- ◆ Associativity (a): Is collision frequent in TLB? (YES)
 - Typically utilize a fully-associative TLB
 - Nowadays, utilize 2~4-way associative TLB

Miss Handling ...

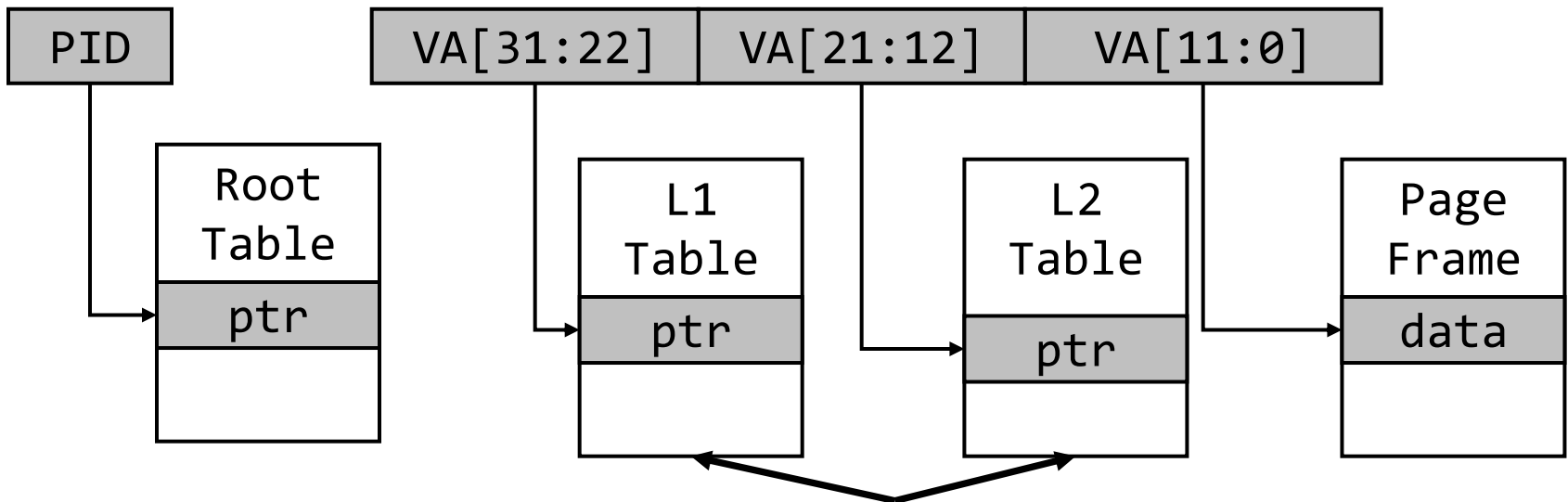
- ◆ Most address translation resolved in ~1 cycle in the TLB
 - L2 TLB takes around < 10 cycles
- ◆ On a TLB miss *miss → DRAM access 3x*
 - Must traverse (walk) the page table for translation using a dedicated hardware
 - Table walk can take 100+ cycles to complete



Mitigate the overhead ...

Bottom-Up Page Table Walk

- ◆ **Key insight:** in top-down page table walk, we played with physical addresses! → why not play with the virtual addresses once more ...



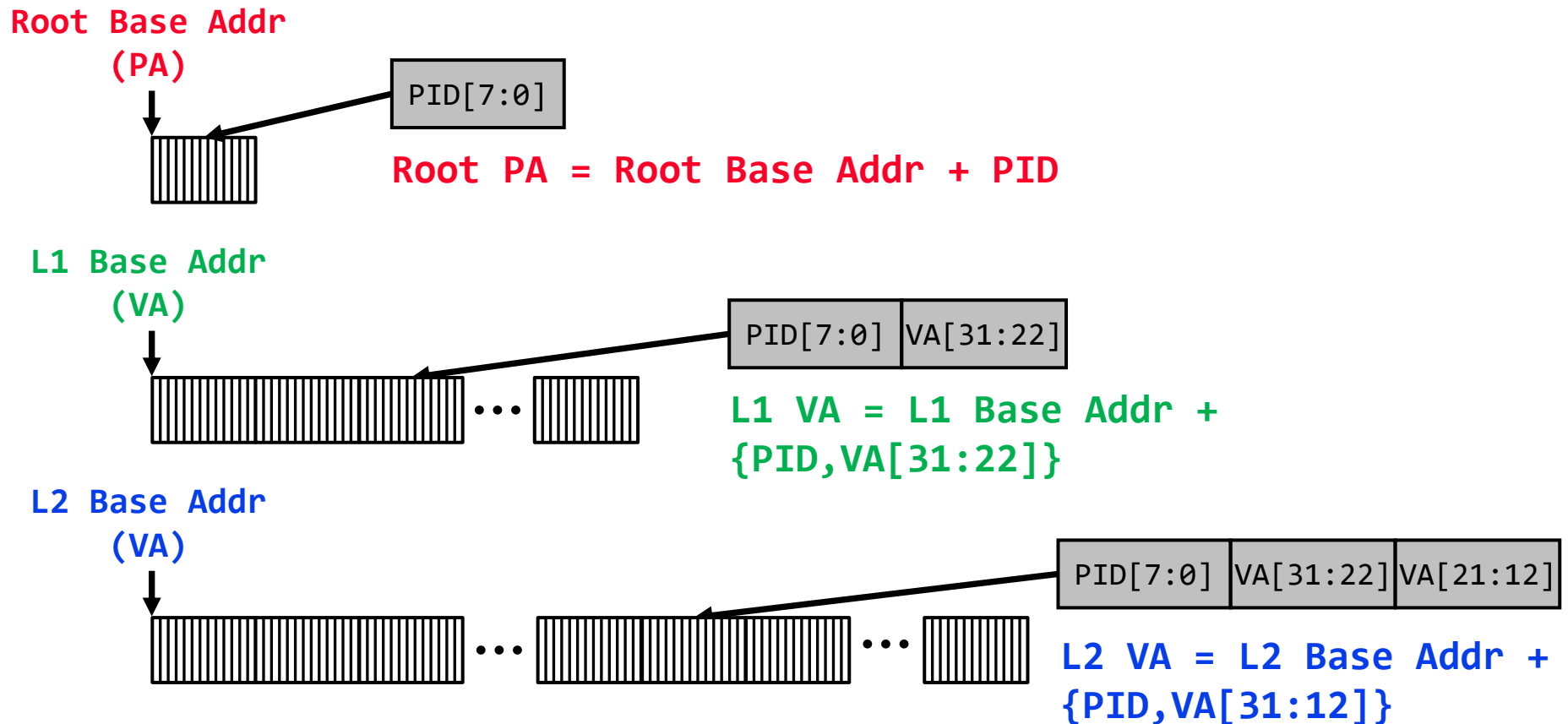
*We do not know the physical addresses of the tables
→ That's why we had to traverse all the way here*

But, what about the virtual addresses?

Mitigate the overhead ...

Bottom-Up Page Table Walk

- ◆ The page table entries are contiguous according to the virtual addresses



L1 VA

Root

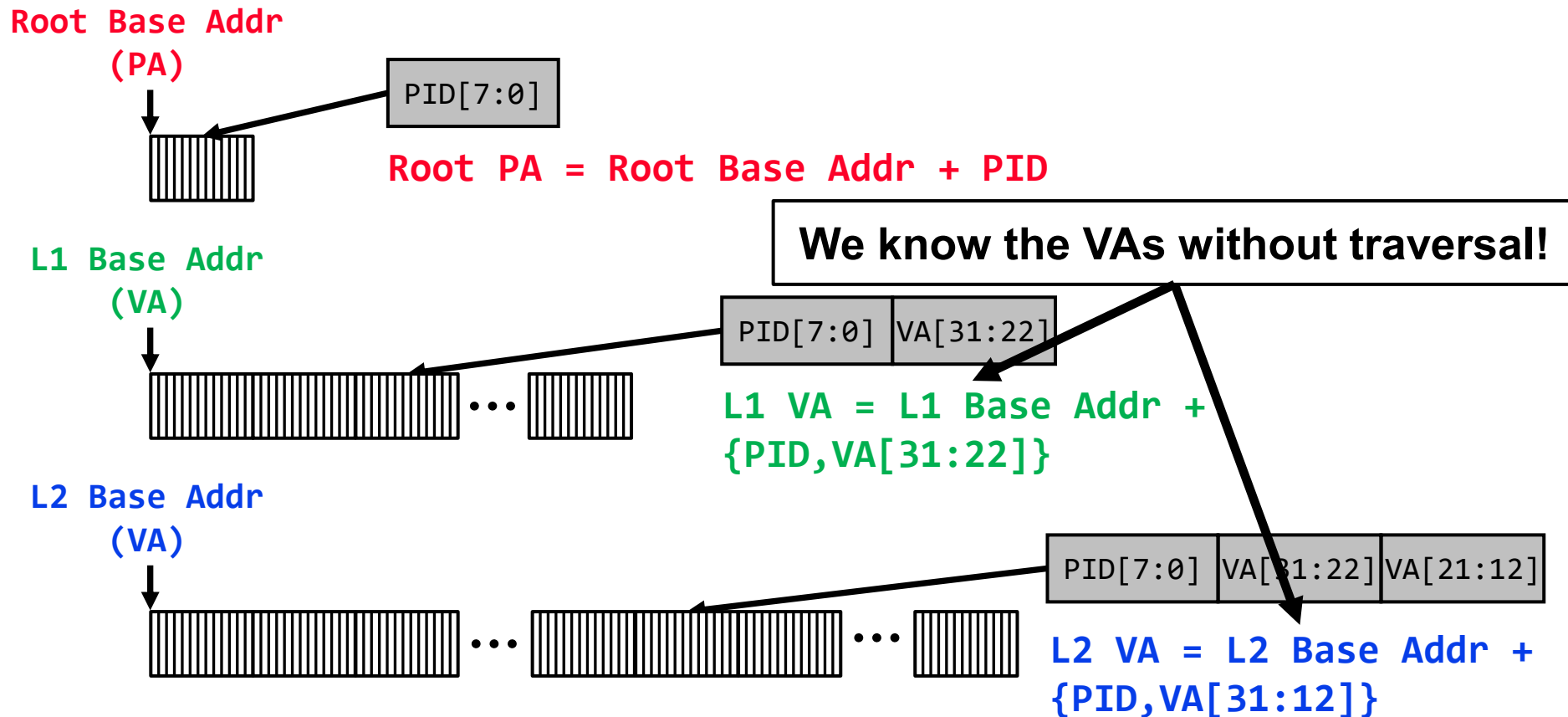
[VA[31:22]]

| |
|-------|
| L1(0) |
| L1(1) |
| L1(2) |
| |

Mitigate the overhead ...

Bottom-Up Page Table Walk

- ◆ The page table entries are contiguous according to the virtual addresses



Mitigate the overhead ...

Bottom-Up Page Table Walk

- ◆ We know the **virtual addresses** (not the physical addresses) of the L1 and L2 page table entries without the hierarchical page table walk
- ◆ Then, why not translate the L2's virtual address into the physical address by accessing the TLB?
 - **If TLB Hit:** we know the physical address of the L2 page table, so we can access the mapping table to obtain the PPN
 - *We obtain PPN with only a single additional memory access (at the cost of only a single TLB access)*
 - **If TLB Miss:** try out again with L1's virtual address!
 - If Hit: *We obtain PPN with only two additional memory accesses (at the cost of two TLB accesses)*
 - If Miss: *you know what will happen!*

What about inverted page table

◆ Baseline:

- Store the VPN along with the next address in the PTE
- Let's assume that
 - $\text{hash}(\text{VPN}) \rightarrow \text{VPN} \% 5$
 - Upon collision: $\text{VPN} += 3$
 - Allocate $\text{VPN}(100)$, $\text{VPN}(200)$, $\text{VPN}(300)$, **$\text{VPN}(101)$**

| idx (== PPN) | VPN | Next |
|--------------|-----|------|
| 0 | 100 | 3 |
| 1 | 300 | - |
| 2 | | |
| 3 | 200 | 1 |
| 4 | | |




| idx (== PPN) | VPN | Next |
|------------------|------------|----------|
| 0 | 100 | 3 |
| 1 (Coll!) | 300 | 4 |
| 2 | | |
| 3 | 200 | 1 |
| 4 | 101 | - |

Inverted page table

- ◆ To avoid collision, we must increase the table size ... (larger than the physical address size)
- ◆ The problem → we must include the PPN to the inverted page table!
- ◆ If there are eight entries (up to five physical table entries)

| idx | PPN | VPN | Next |
|-----|-----|-----|------|
| 0 | 0 | 200 | - |
| 1 | 3 | 101 | - |
| 2 | | | |
| 3 | | | |
| 4 | 1 | 100 | 7 |
| 5 | | | |
| 6 | | | |
| 7 | 2 | 300 | - |

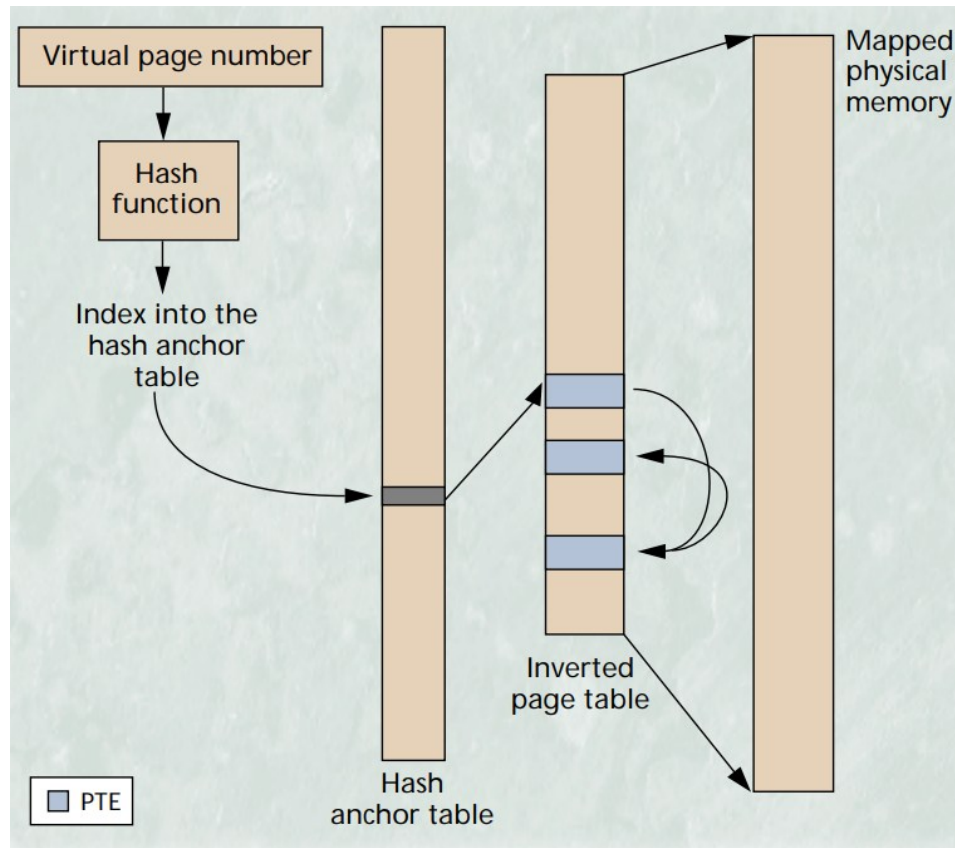


*Reducing collision
requires increasing the
number of PTE entries*

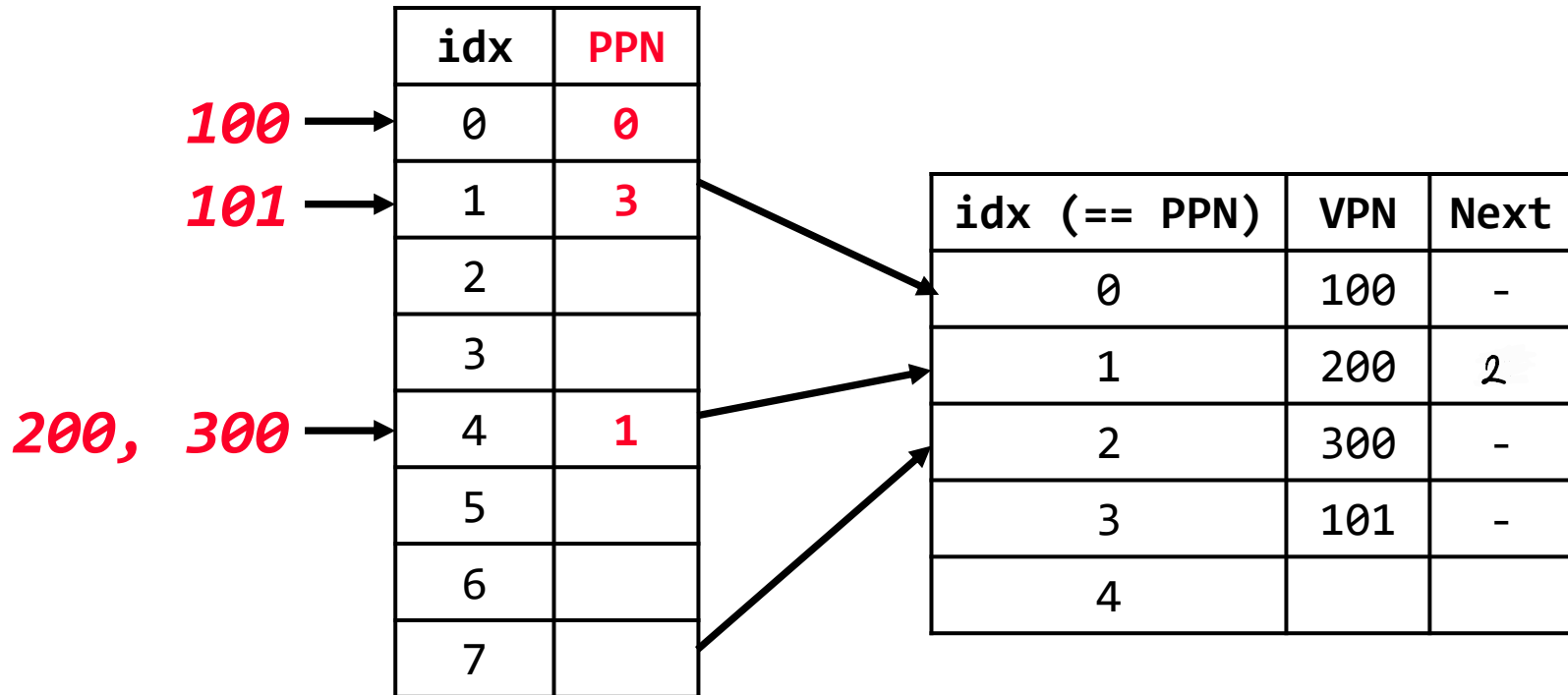
**→ But, the PTE entry
should keep “a large
number of bits” →
PPN, VPN, Next ...**

Hash anchor table!

- ◆ We may utilize the hash anchor table to perform a single level of indirection!



Hash anchor table!



*We can increase the hash space **with less overhead** (hash anchor table only keeps the idx → instead of the VPN, PPN, next, ... as in the original hash table)*

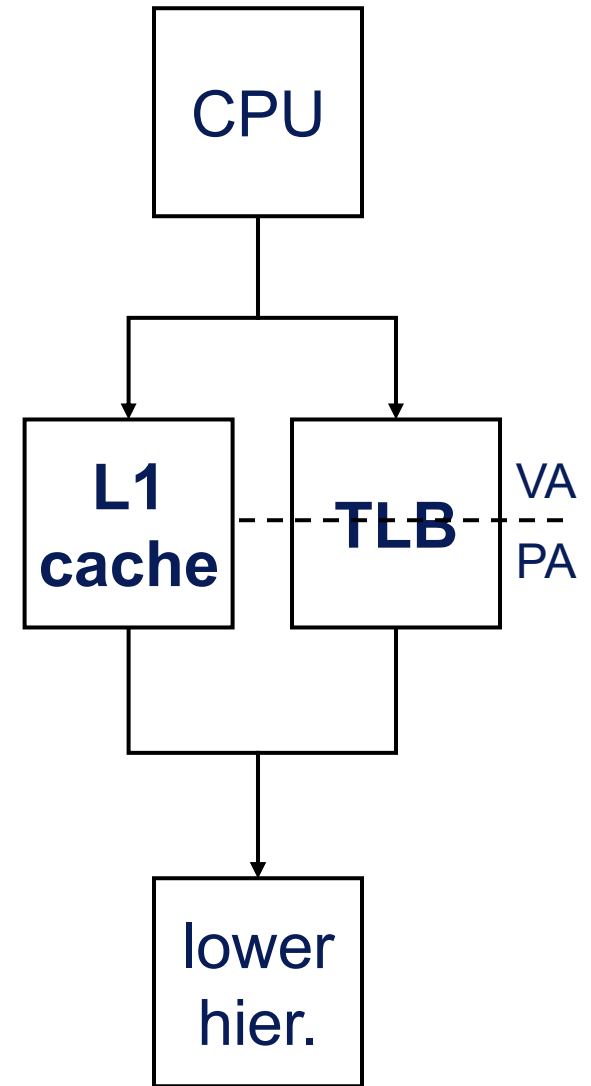
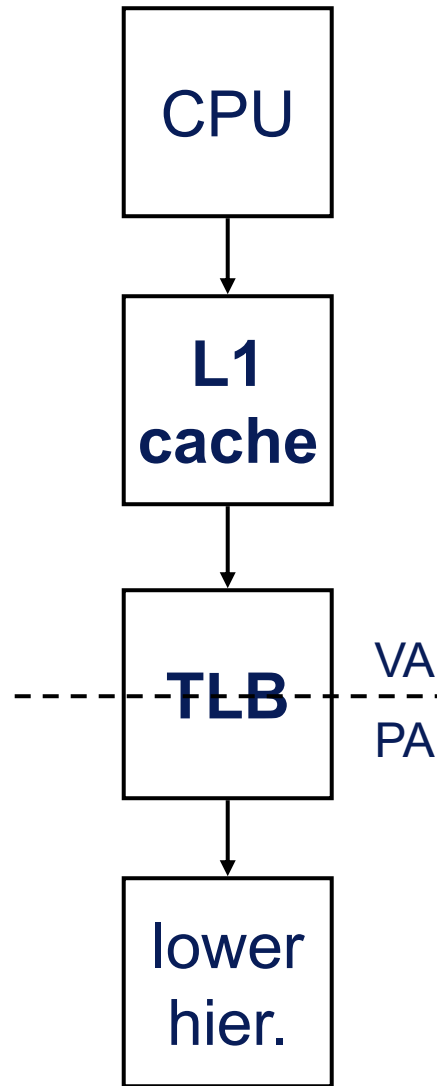
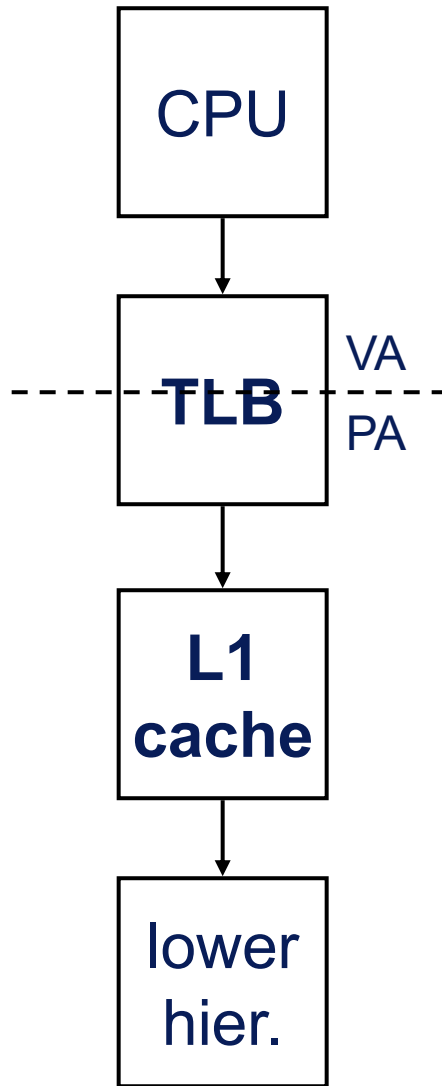
Modern Translation Mitigation Techniques!

- ◆ **Option #1:** Utilize an L2 TLB to minimize the TLB miss rate!
 - Separate L1 I-TLB + L1 D-TLB, but unified L2 TLB
- ◆ **Option #2:** Hardware page table walker
 - Instead of relying on the OS to handle the fault, use the hardware to support page table walk! → Reduces context switching & SW handling overhead!
- ◆ **Option #3:** Page walk cache
 - Cache this also ... keep entries of frequently accessed page table entries
- ◆ **Option #4:** Heterogeneous page tables
 - There are different page sizes (e.g., 4 KB, 2 MB, 1 GB)

After the page table walk

- ◆ We eventually find the PTE (or not if it does not exist)
 - **Case #1 (PTE valid & page in memory):** replace TLB with a new PTE and continue
 - **Case #2 (PTE valid & page in disk):** trigger “page fault” exception to initiate kernel handler for demand paging and update the page table
 - **Case #3 (PTE invalid or not found):** trigger “page fault” → and the fault may be escalated to “segmentation fault” depending on why the page fault was triggered!

How do VM and cache interact?



Physical cache

Virtual (L1) cache

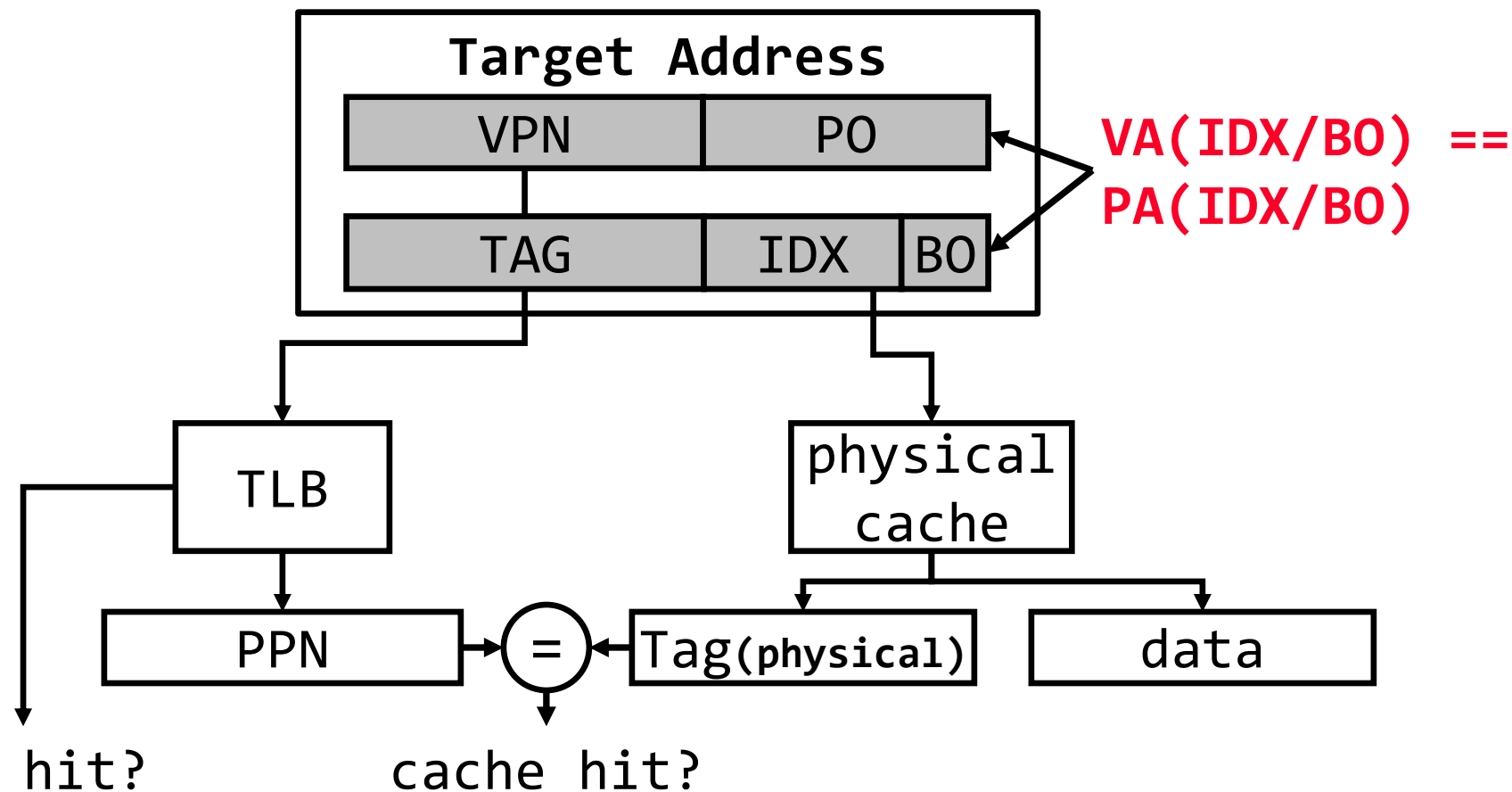
Hybrid!

L1 caching and VM

- ◆ If you use a physical cache (TLB → Cache):
 - Even in the best case, the latency is → **TLB hit** + L1 cache hit
 - Reducing the L1 latency is highly important!
- ◆ Ok ... what about virtual cache (Cache → TLB):
 - **Benefit:** We can reduce the translation overhead as we can translate the address only upon a cache miss!
 - Highly effective if TLB hit time \gg L1 cache hit time (not true)
 - **Problem #1 (Homonyms):** Same EAs (in different processes) point to different PAs
 - **Solution #1:** Flush virtual cache between context (cannot support SMT)
 - **Solution #2:** Include PID in the cache tag (additional cache overhead)
 - **Problem #2 (Synonyms):** Different EAs (in different processes) point to the same PA (sharing addresses)
 - A PA could be cached twice under different EAs.
 - Updating one cached copy would not reflect the other copy
 - **Solution:** Make sure that synonyms can't coexist in the cache

Combining both approaches?

- ◆ Hybrid: Virtually-indexed and physically-tagged cache
- ◆ Key insight → the size of the L1 cache is small
 - The cache index bits come only from page offset!



Combining both approaches?

- ◆ Hybrid: Virtually-indexed and physically-tagged cache
- ◆ Key insight → the size of the L1 cache is small
 - The cache index bits come only from page offset!

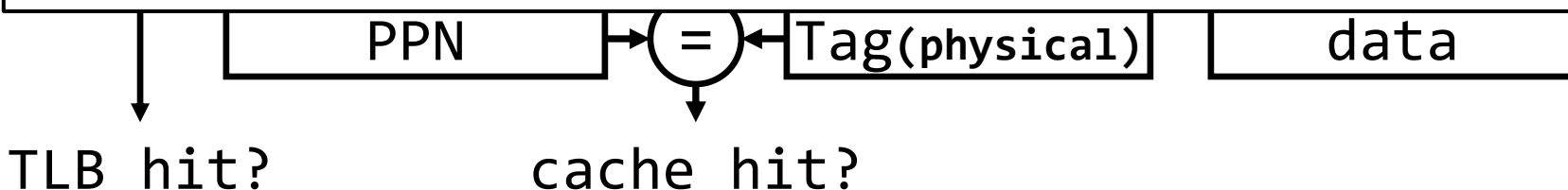


We use physical tag (instead of virtual tag)

- (1) Homonym X: We know which process the data belongs to
- (2) Synonym X: We do not duplicate data (that have different VAs, but the same PA)

Physical tag은 같은 → 서로 다른 VA가 서로 다른 virtual memory에 저장될 수 있다.

+ It enables parallel translation (FAST!) ← 장점: 빠르다



Using alias!

23 page는 a 00 PO 에
 → L1에 00 PO에 Index로 저장
 이후 같은 Request
 → L1에 01 PO가 들어
 → 하지만 L1에 같은 해

이런 경우 synonym이 생
 이라고 된다고

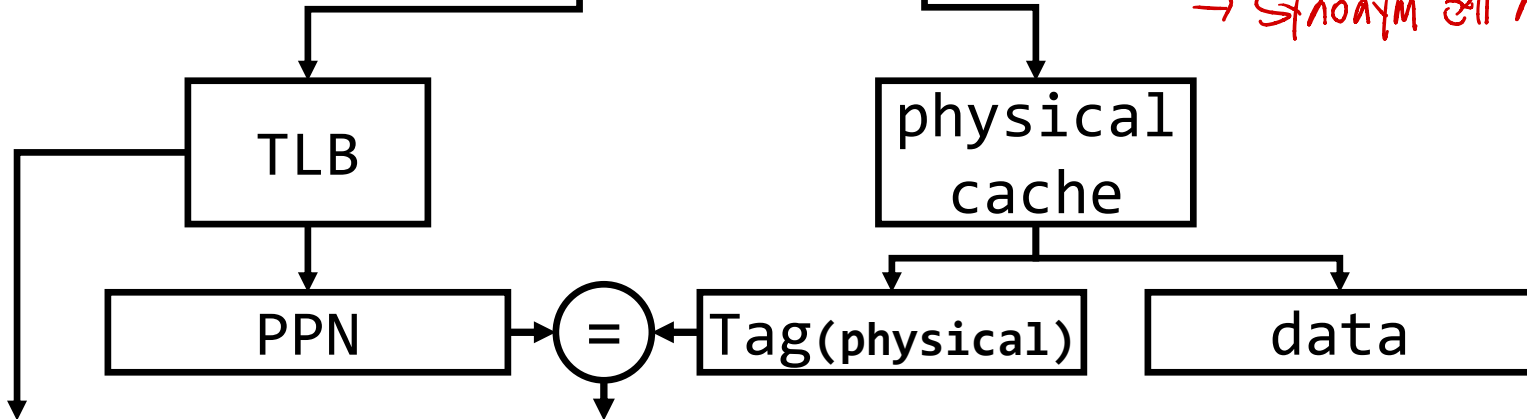
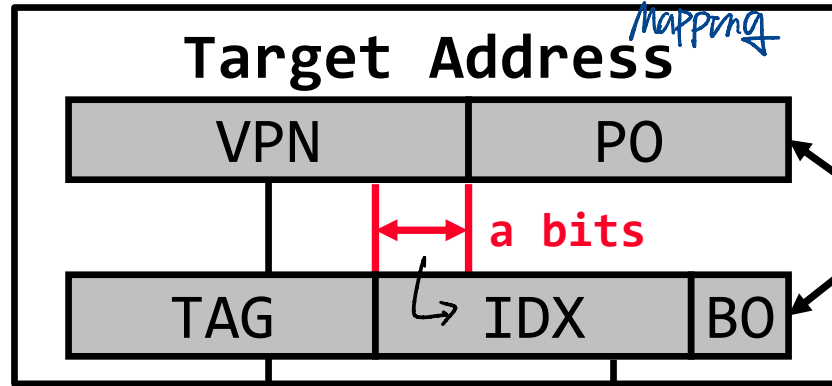
- What happens if L1 cache size is too large →
 - $\text{BITS}(\text{IDX}) + \text{BITS}(\text{BO}) > \text{BITS}(\text{PO})$
- We use a part of the virtual address (i.e., $\text{VPN}[a-1:0]$) to index the physical address

같은 PA
 Mapping

| | | |
|---|----|----|
| a | 00 | PO |
| y | 01 | PO |

→ synonym 생
 $\text{VA}(\text{IDX}/\text{BO}) == \text{PA}(\text{IDX}/\text{BO})$

a bits를 뺀다
 → synonym에 재발



TLB hit?

cache hit?

How to support this?

- ◆ Homonym?? → No, the memory still has a physical tag
- ◆ Synonym?? → Potentially yes, if the two virtual addresses differ by only a -bit LSBs
- ◆ **Option #1:** We can increase the cache associativity to reduce the index bits → Not so good
- ◆ **Option #2:** Enforce mapping to ensure no synonym occurs (e.g., enforce $\text{VPN}[a-1:0] == \text{PPN}[a-1:0]$)
- ◆ **Option #3:** Use L2 cache to detect synonyms
 - Append $\text{VPN}[a-1:0]$ to the tag of L2 (L2 can now detect synonym)
 - Therefore, it can detect synonyms and flush L1 entries

Synonym Resolving!

◆ Example → **VA** and **VB** map to the same physical address (synonym problem)

- **Main assumptions:**

- There is an alias 'a' of two bits
- L1 is VIPT (virtually indexed and physically tagged) and L2 is a physical cache (but, has VPN[1:0] appended to the tag!)
- Suppose **VA** is accessed first and blocks are allocated in L1 and L2 cache

L1 Hit Miss? L2 Hit → 0 (VA, Tag) 다르면 Synonym 문제 발생

- **VB** is accessed and results in L1 cache miss (as it indexes to a different block!)
- **VB** translates to PA and goes to the same block as **VA** in “L2”
- Tag comparison fails as ($VA[1:0] \neq VB[1:0]$)
- L2 detects that a synonym (VA) is cached in L1 (this is required!)
→ Evict VA's entry in L1 before VB is allowed to be refilled in L1

TLB Management

- ◆ There are two ways to handle the TLB miss!
- ◆ **Option #1: HW-managed solution**
 - It is much faster, but suffers from limited flexibility
- ◆ **Option #2: SW-managed solution**
 - It is much slower (it may take 10~100 instructions to support exception!), but highly flexible.

Page table implementation

◆ Hierarchical?

- Can reduce the size of table if locality exists
- Multi-level page table walk on TLB miss
 - Maybe, bottom-up lookup?
 - Originally implemented in MIPS

◆ Hashed?

- The size grows linearly with the size of physical memory
- Possible hash collisions require a back-up page table

Dive into the examples!

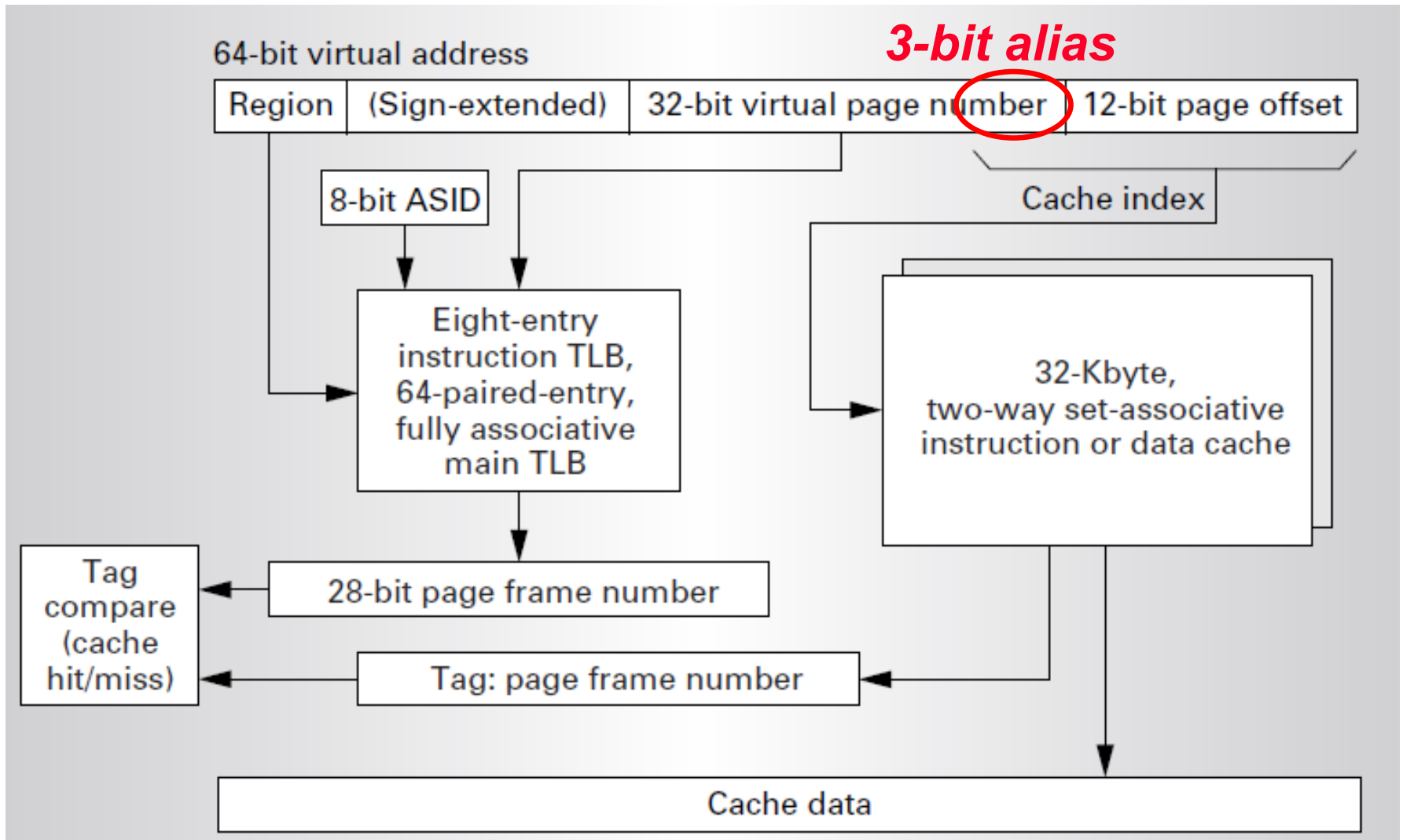
MIPS R10K

- ◆ MIPS R10K utilizes a 64-bit virtual address space
- ◆ Requires an 8-bit ASID (or PID) to distinguish between processes!
- ◆ R10K supports up to 40-bit physical address space
 - If the installed DRAM is smaller than 2^{40} byte \rightarrow the OS will allocate only a portion of the physical address space
 - If the installed DRAM is larger than 2^{40} byte \rightarrow cannot fully access the DRAM!
- ◆ Summary:
 - ***64-bit VA (with 8-bit ASID) to 40-bit PA***

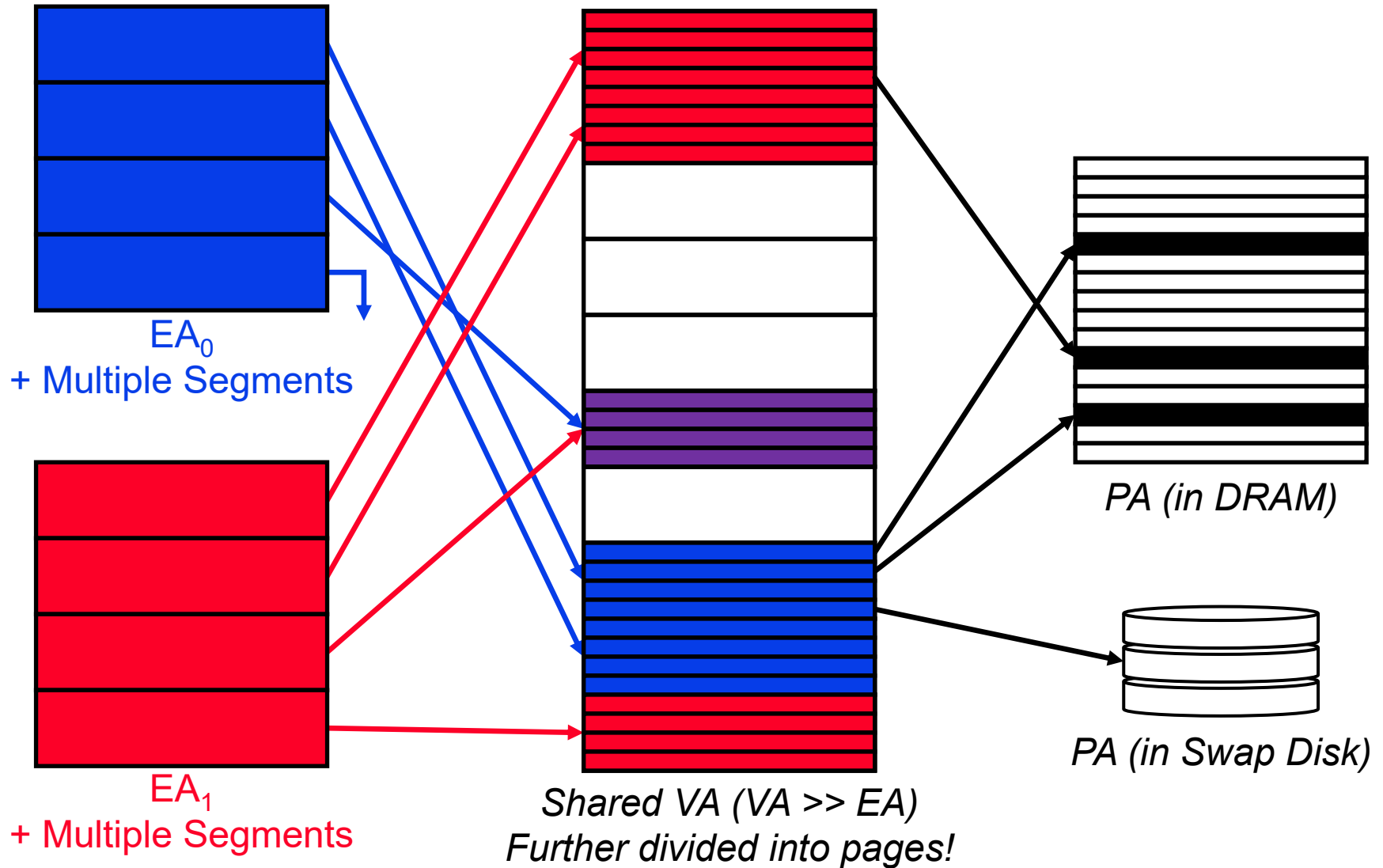
TLB in MIPS R10K

- ◆ 64-entry fully associative unified TLB
- ◆ VIPT (virtually indexed physically tagged) + 3-bit alias
- ◆ Paired (each entry maps 2 consecutive VPNs) → similar to block in cache!
- ◆ TLB miss handled in SW

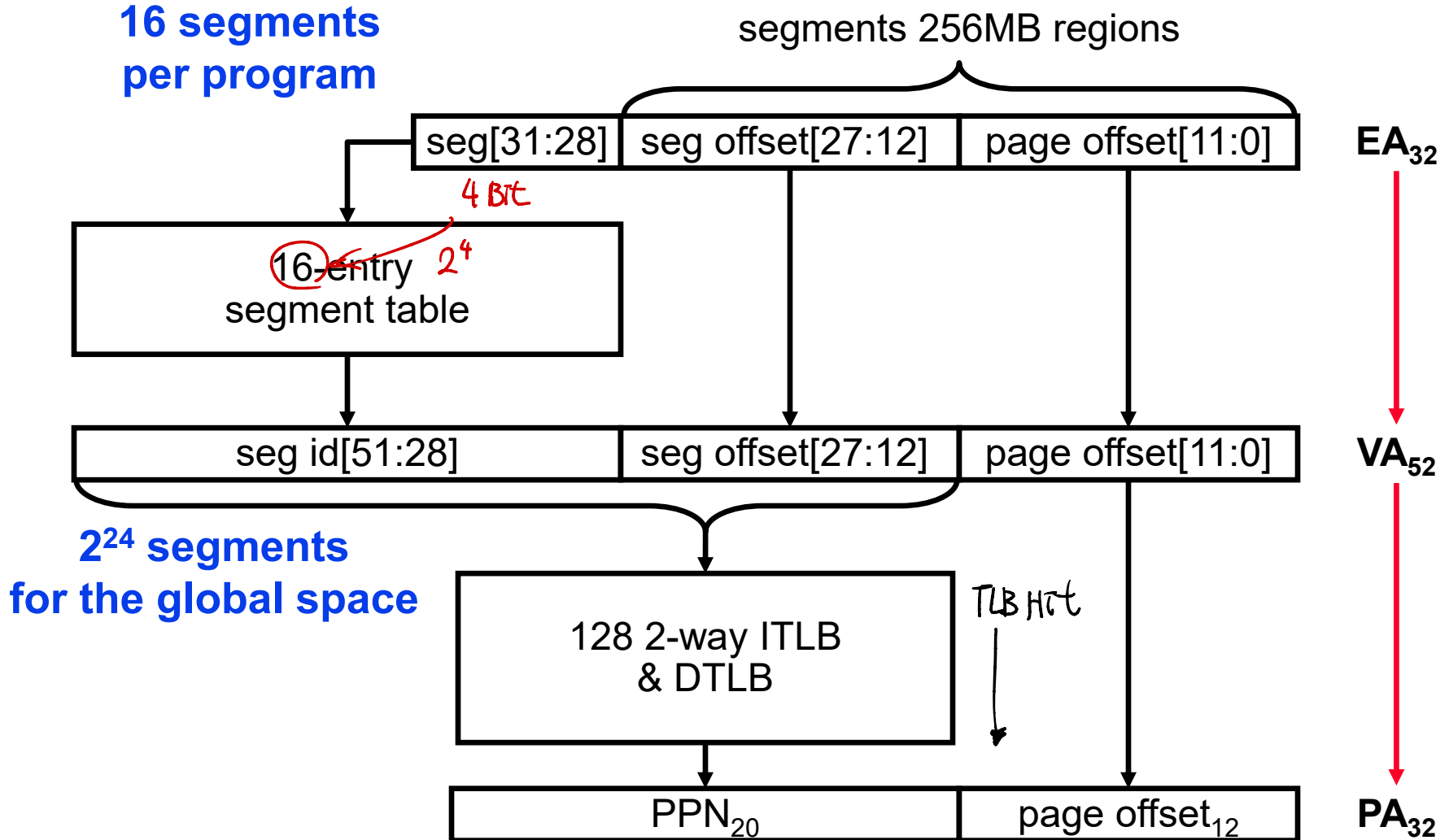
MIPS R10K



IBM PowerPC

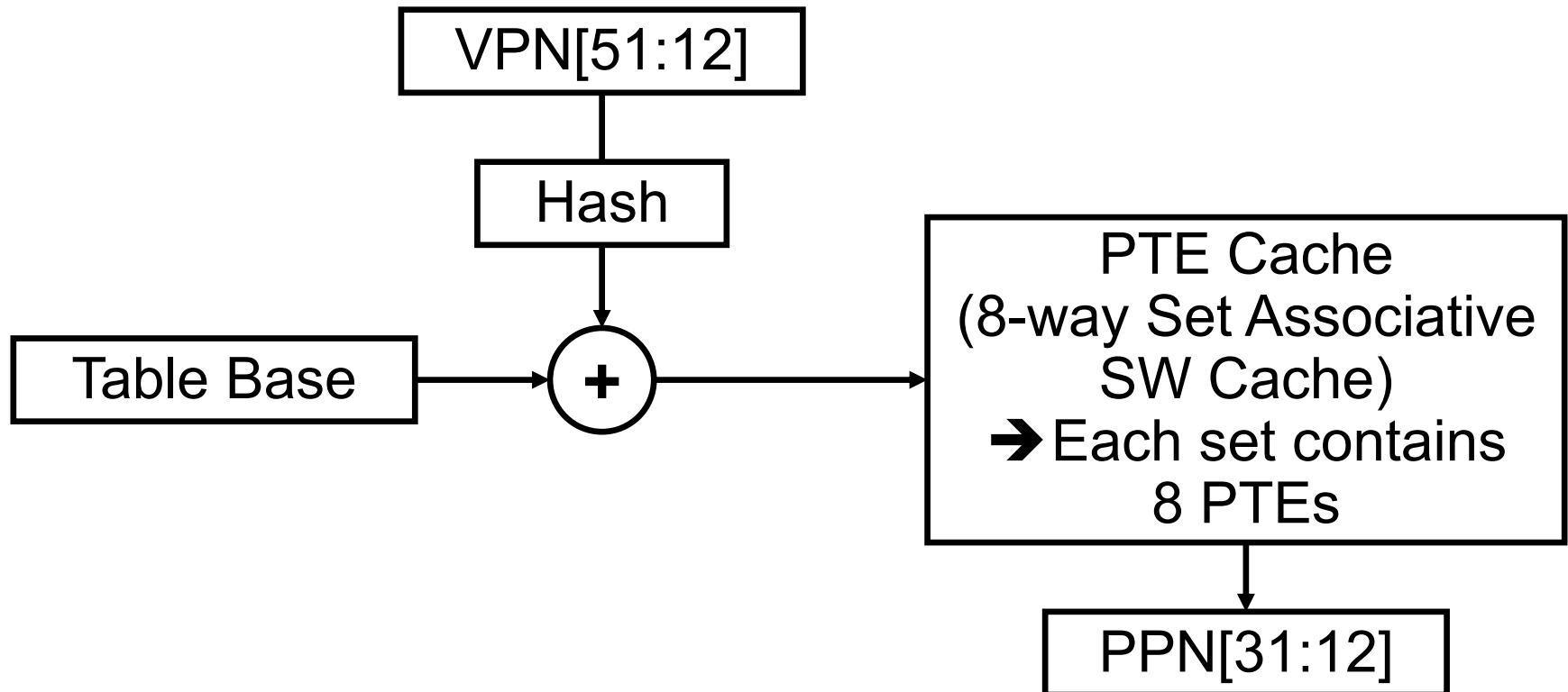


IBM Power PC



Page Table in PowerPC

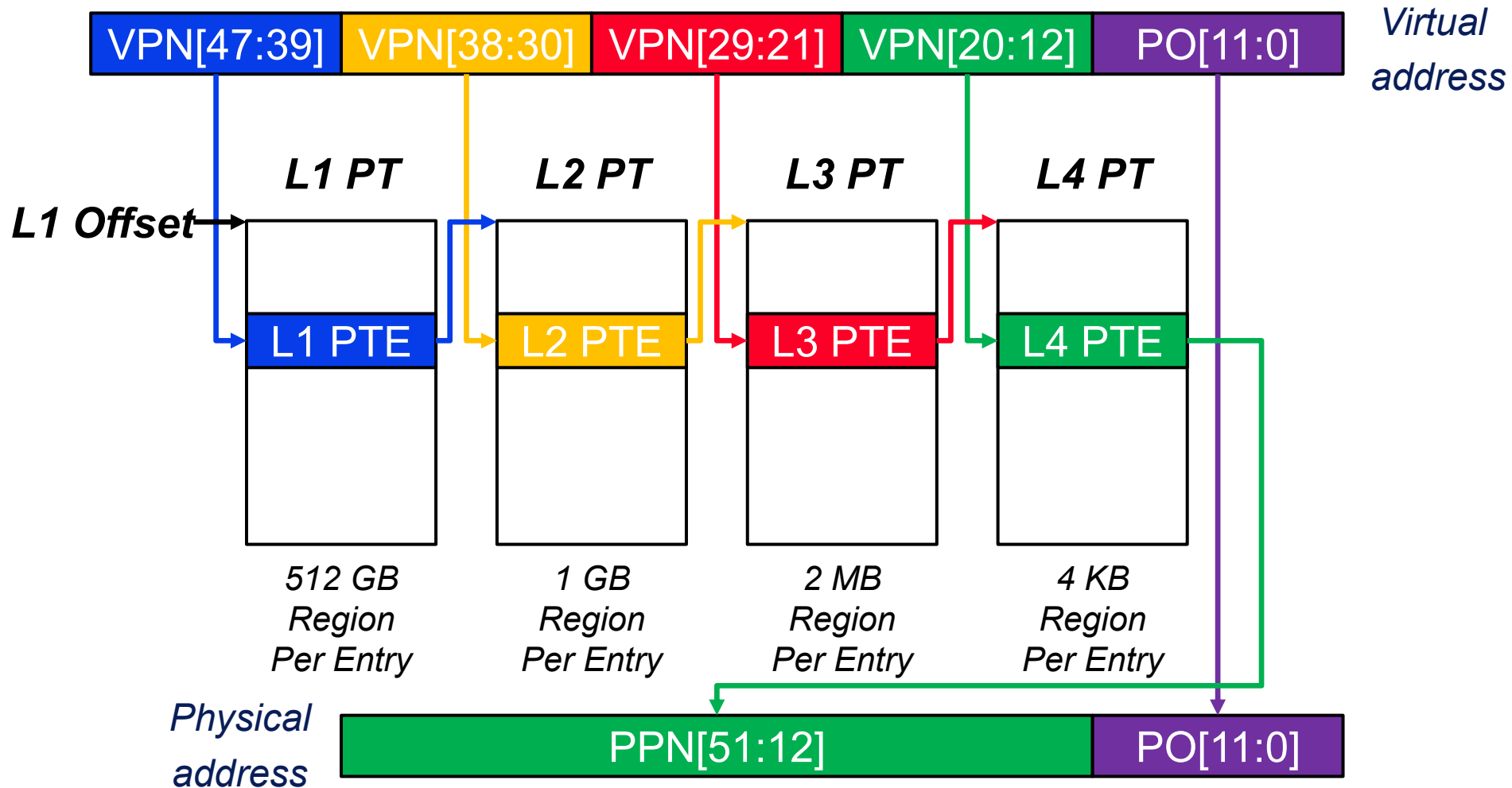
- IBM power PTE 2.0*
- ◆ **Inverted (Hashed) Page Table** with a SW-controlled cache to keep the frequently accessed PTE entries!



Intel x86

- ◆ space, PowerPC, there exists a two-level address translation w/ segments for naming + protection
 - 48-bit effective address → 16-bit segment num + 32-bit segment offset
 - **#1**: 48-bit effective address is translated into the 32-bit virtual address
 - **#2**: 32-bit virtual address is translated into the 32-bit physical address (before Pentium Pro) vs. 36-bit (from Pentium Pro and before x86-64)
48 bit → 32 bit → 32 bit
- ◆ If it keeps only a 32-bit virtual address space, how can we simultaneously execute multiple processors?
 - It would exceed the 2^{32} -memory space
- ◆ x86 keeps a separate VA space for each process (time multiplex VA) → requires separate mapping table for each process
 - Load different mapping table & flush TLB on context switch ...

i7 (x86-64)

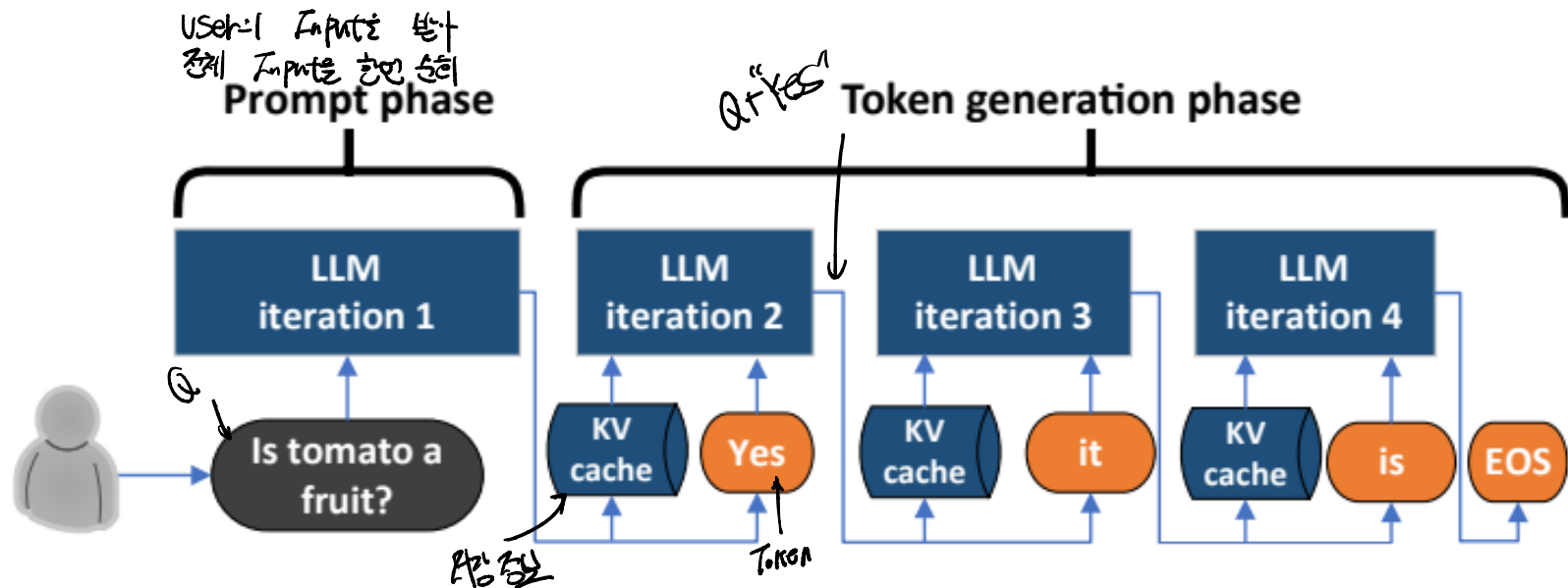


48-bit virtual address to 52-bit physical address translation

➔ Also, separate VA for each process (requires TLB flushing ... or TLB with ASID)

Paged Attention (SOSP'23)

- ◆ Do you know an autoregressive LLM?
- ◆ Note that, I'm making a simplified approximation!



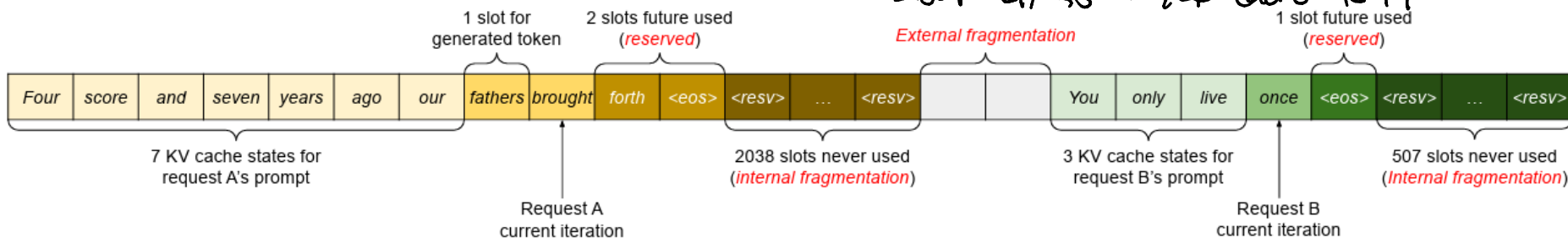
Each token is generated in a step-by-step manner

Step #1. Yes = F(Is tomato a fruit)
Step #2. it = F(Is tomato a fruit, Yes)
Step #3. is = F(Is tomato a fruit, Yes, it)

We need to store the output history to evaluate the next output

Paged Attention (SOSP'23)

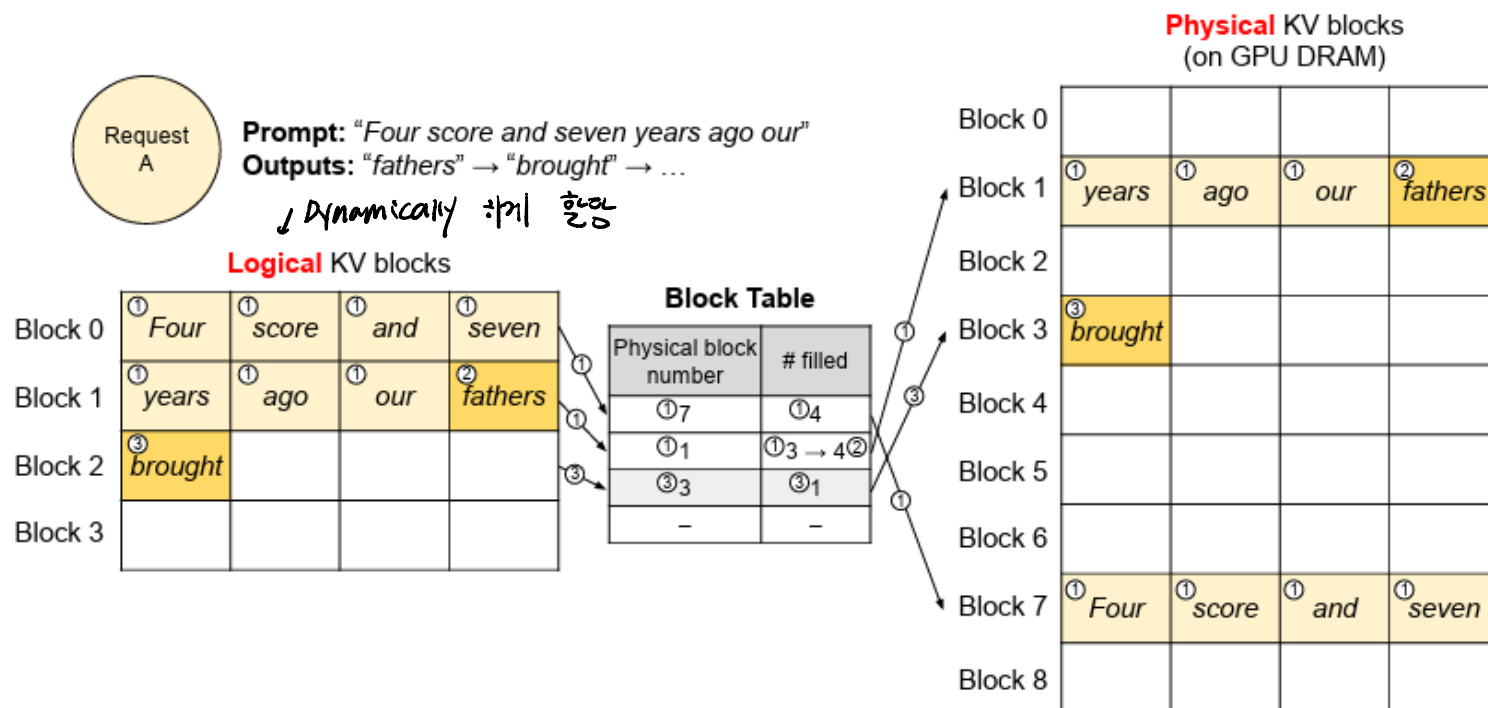
- ◆ **Problem:** we do not know how much memory space is required in advance!
- ◆ **Existing work:** allocate the maximum possible space (e.g., 2048 output tokens)



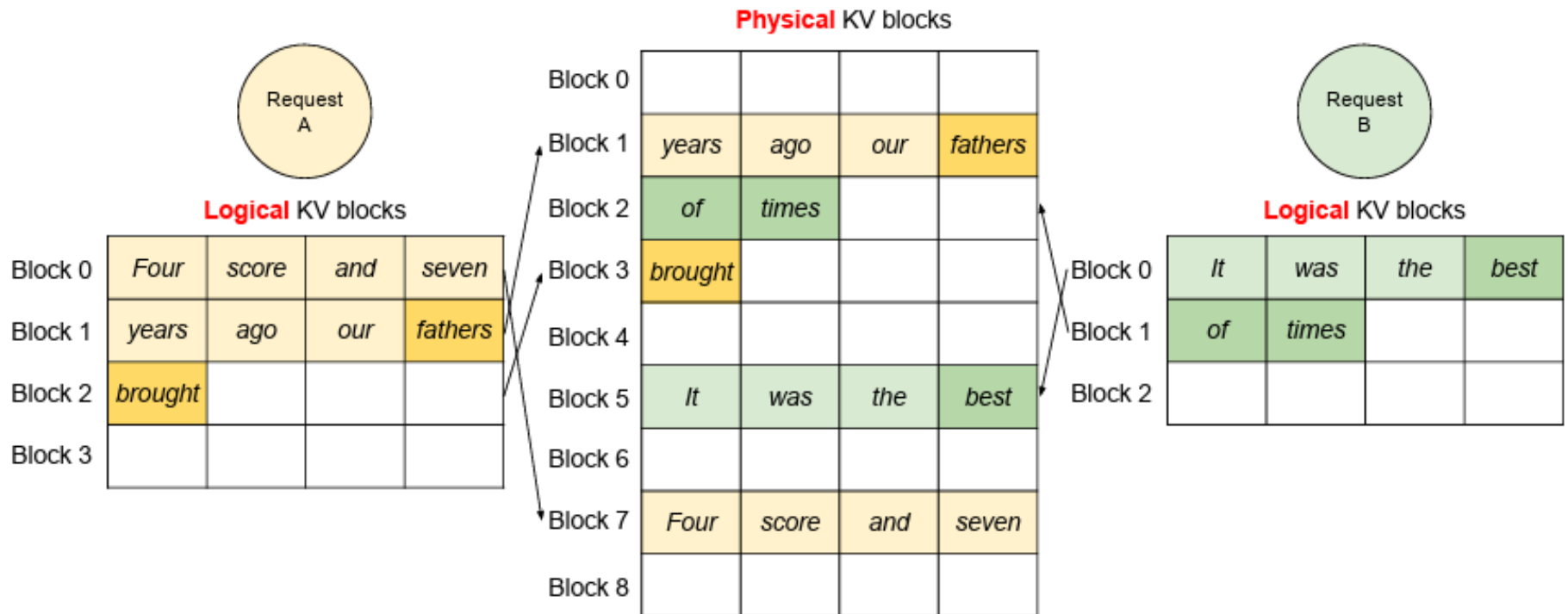
- ◆ This makes **a large memory space left empty!** (as the answer is typically short ...)
 - Especially, when there are multiple requests!

Paged Attention (SOSP'23)

- ◆ **Solution:** you can guess by now ...
- ◆ We see exactly the same problem you saw in the virtual memory! → There are separate **logical KV blocks** and **physical KV blocks**
 - There is a block table to translate the logical to physical blocks



Paged Attention (SOSP'23)



Question?

Announcements:

Reading: finish reading P&H Ch.5

Handouts: none