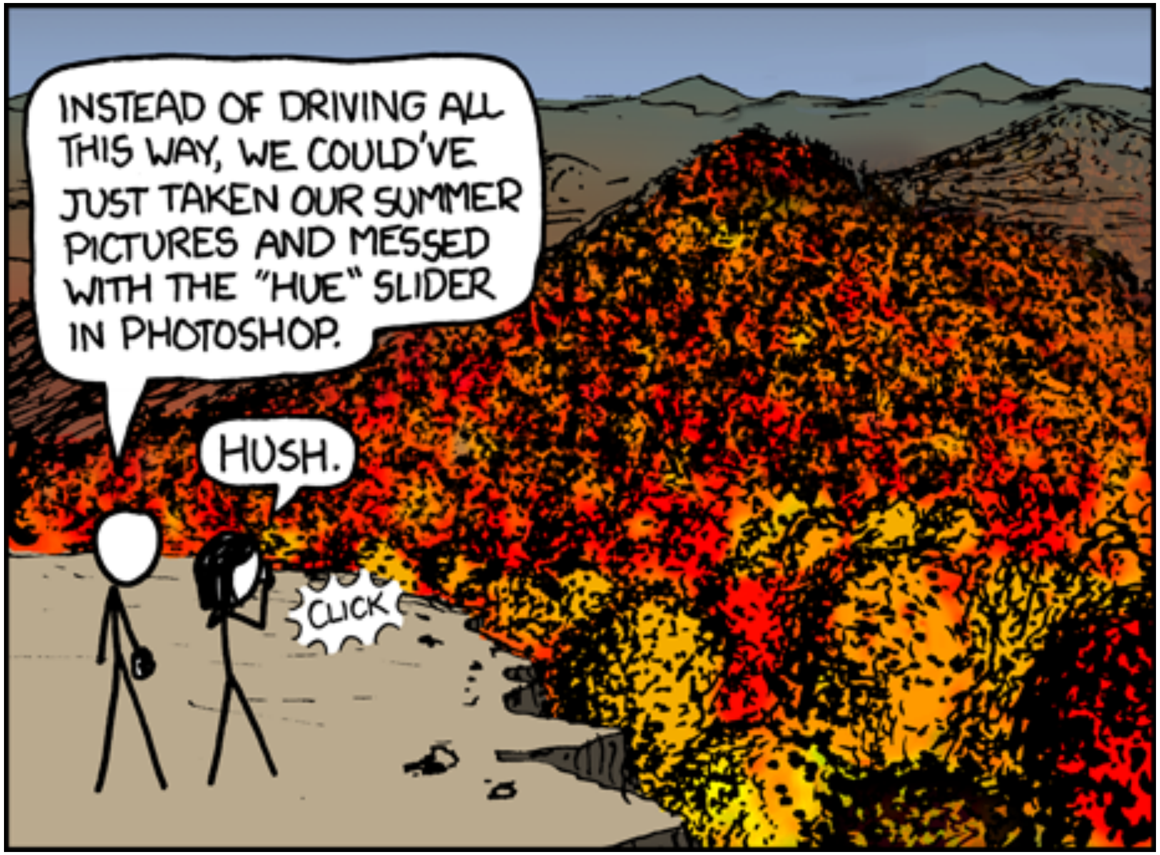
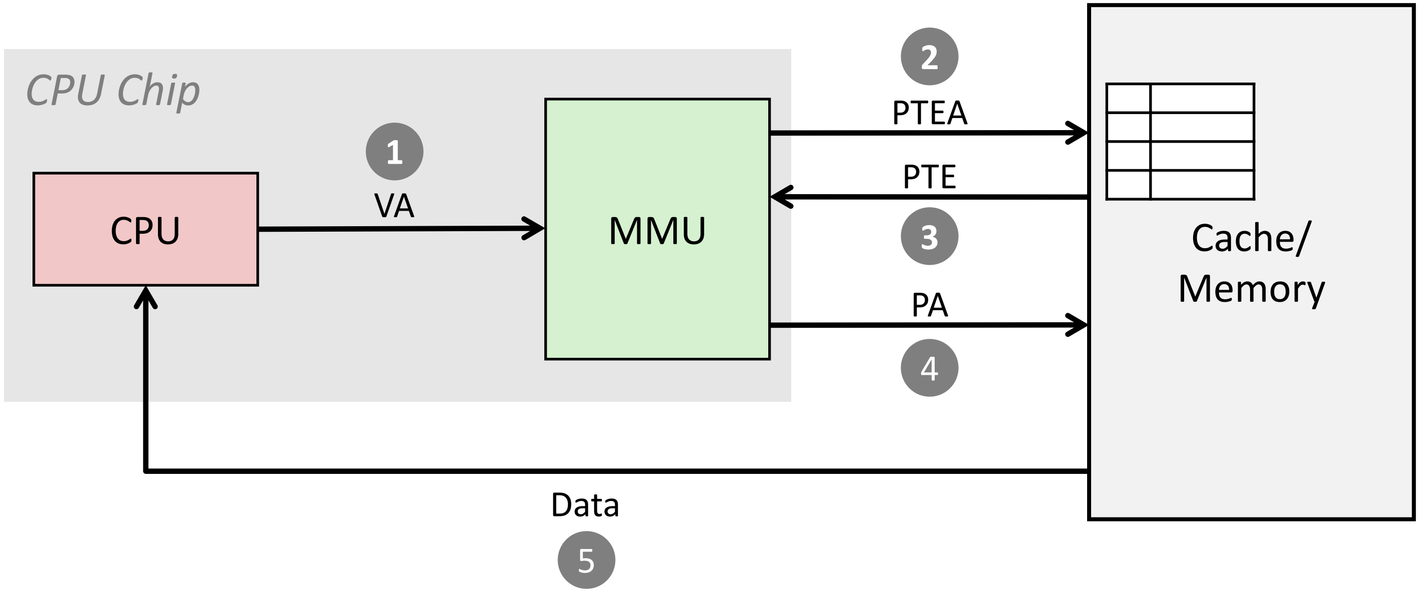
Virtual Memory III



https://xkcd.com/648/

Address Translation: Page Hit



1) Processor sends *virtual* address to MMU (*memory management unit*)

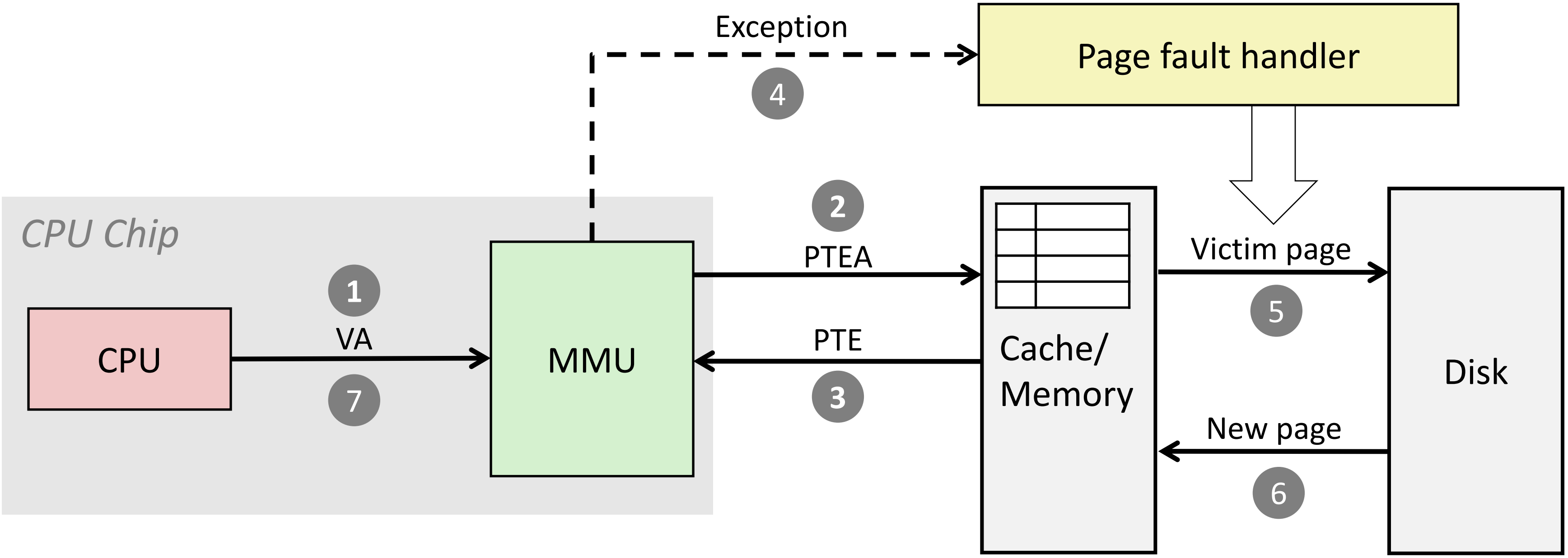
2-3) MMU fetches PTE from page table in cache/memory
(Uses PTBR to find beginning of page table for current process)

4) MMU sends *physical* address to cache/memory requesting data

5) Cache/memory sends data to processor

VA = Virtual Address
PTEA = Page Table Entry Address
PTE= Page Table Entry
PA = Physical Address
Data = Contents of memory stored at VA originally requested by CPU

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in cache/memory

4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

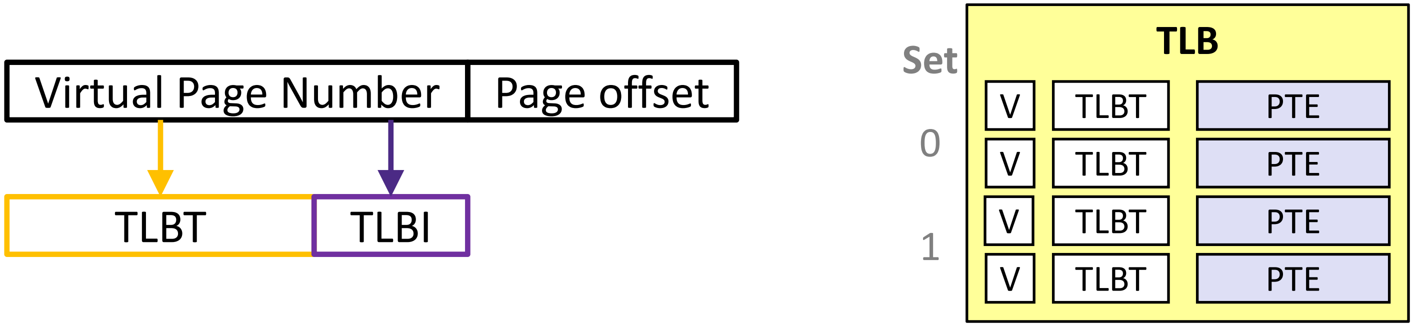
Hmm… Translation Sounds Slow

- The MMU accesses memory *twice*: once to get the PTE for translation, and then again for the actual memory request
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

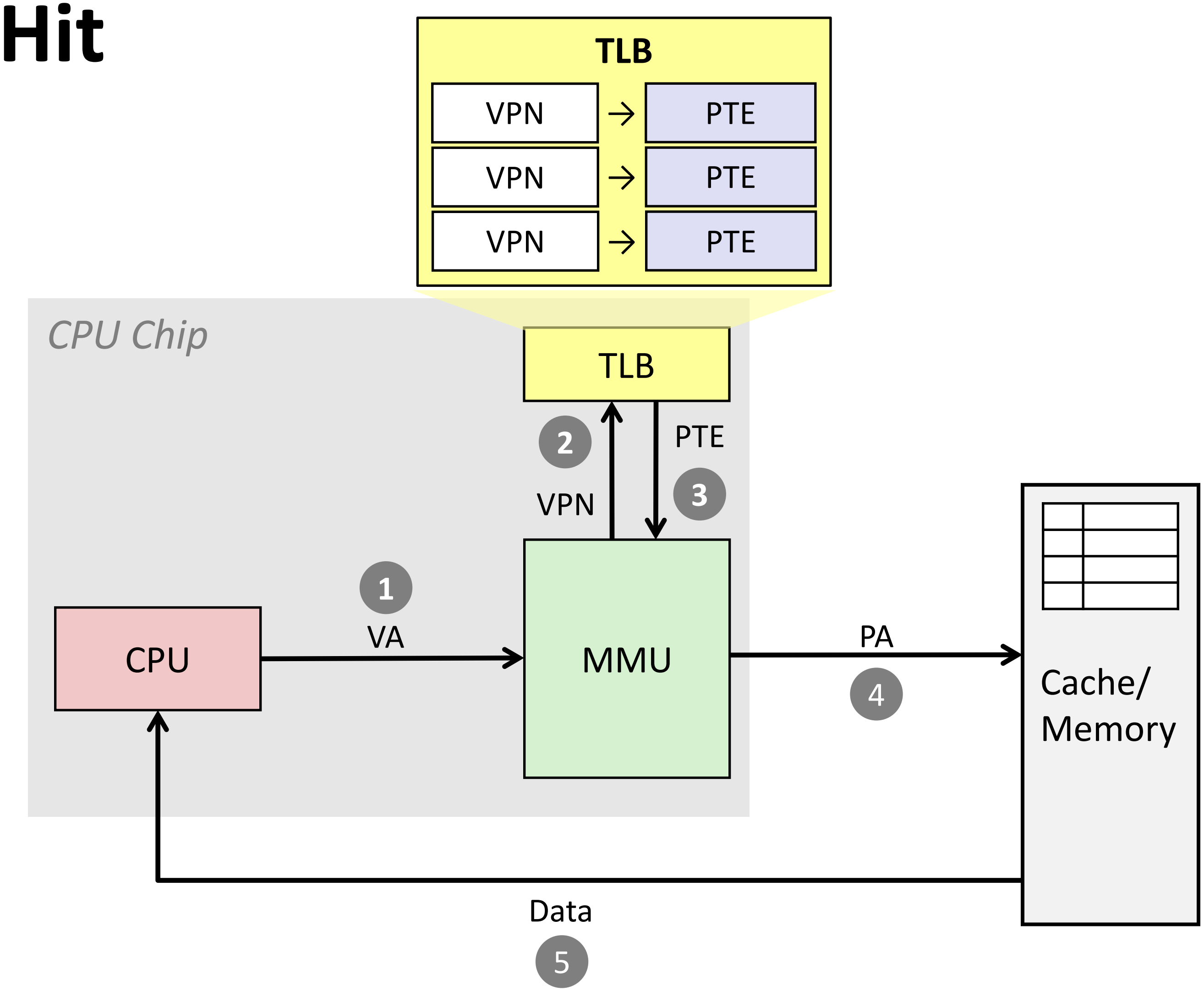
- *What can we do to make this faster?*
  - **Solution:** add another cache! 🎉

Speeding up Translation with a TLB

- *Translation Lookaside Buffer* (TLB):
  - Small hardware cache in MMU
    - Split VPN into **TLB Tag** and **TLB Index** based on # of sets in TLB
  - Maps virtual page numbers to physical page numbers
  - Stores *page table entries* for a small number of pages
    - Modern Intel processors have 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory

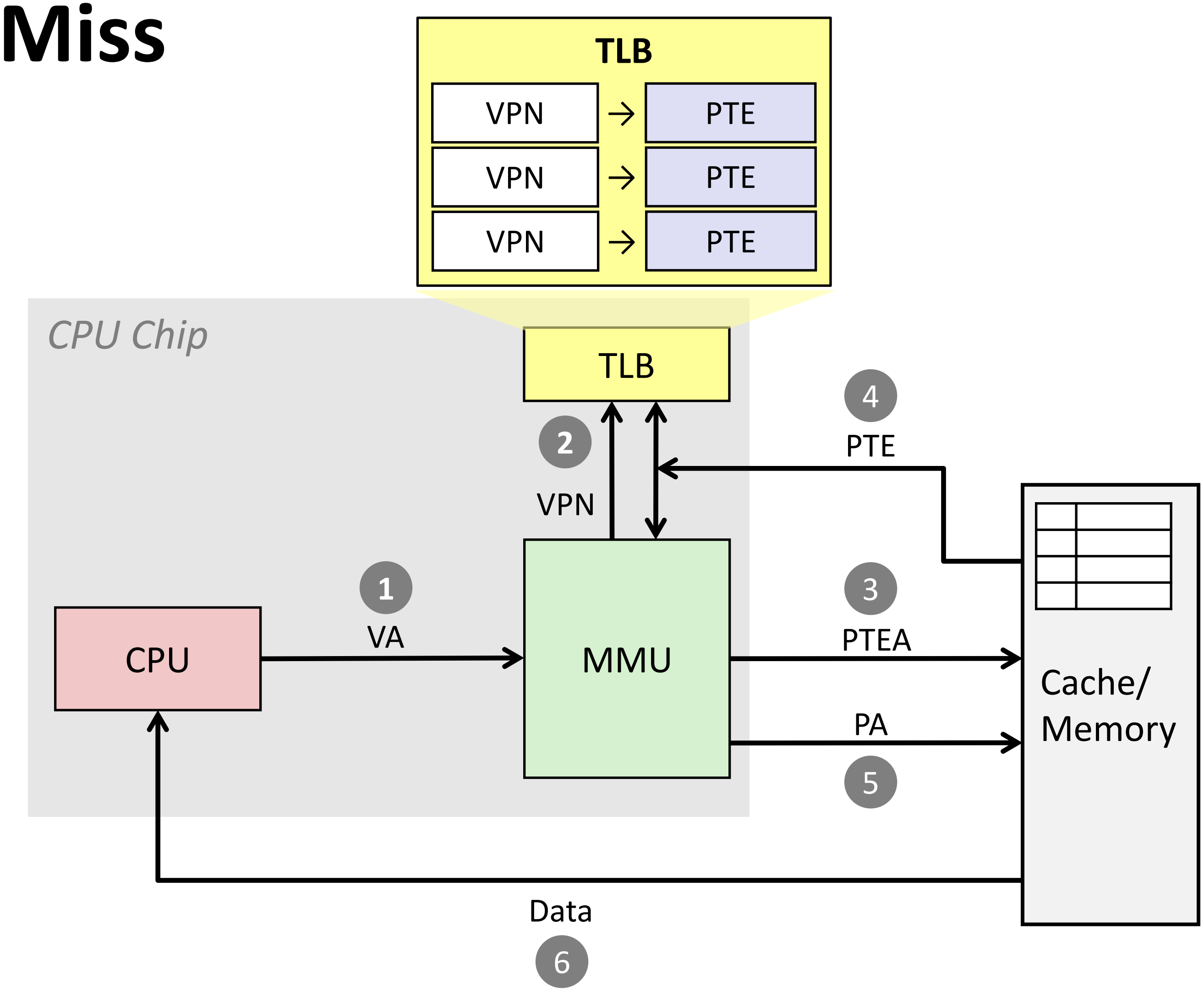


TLB Hit



- A TLB hit eliminates a memory access!

TLB Miss

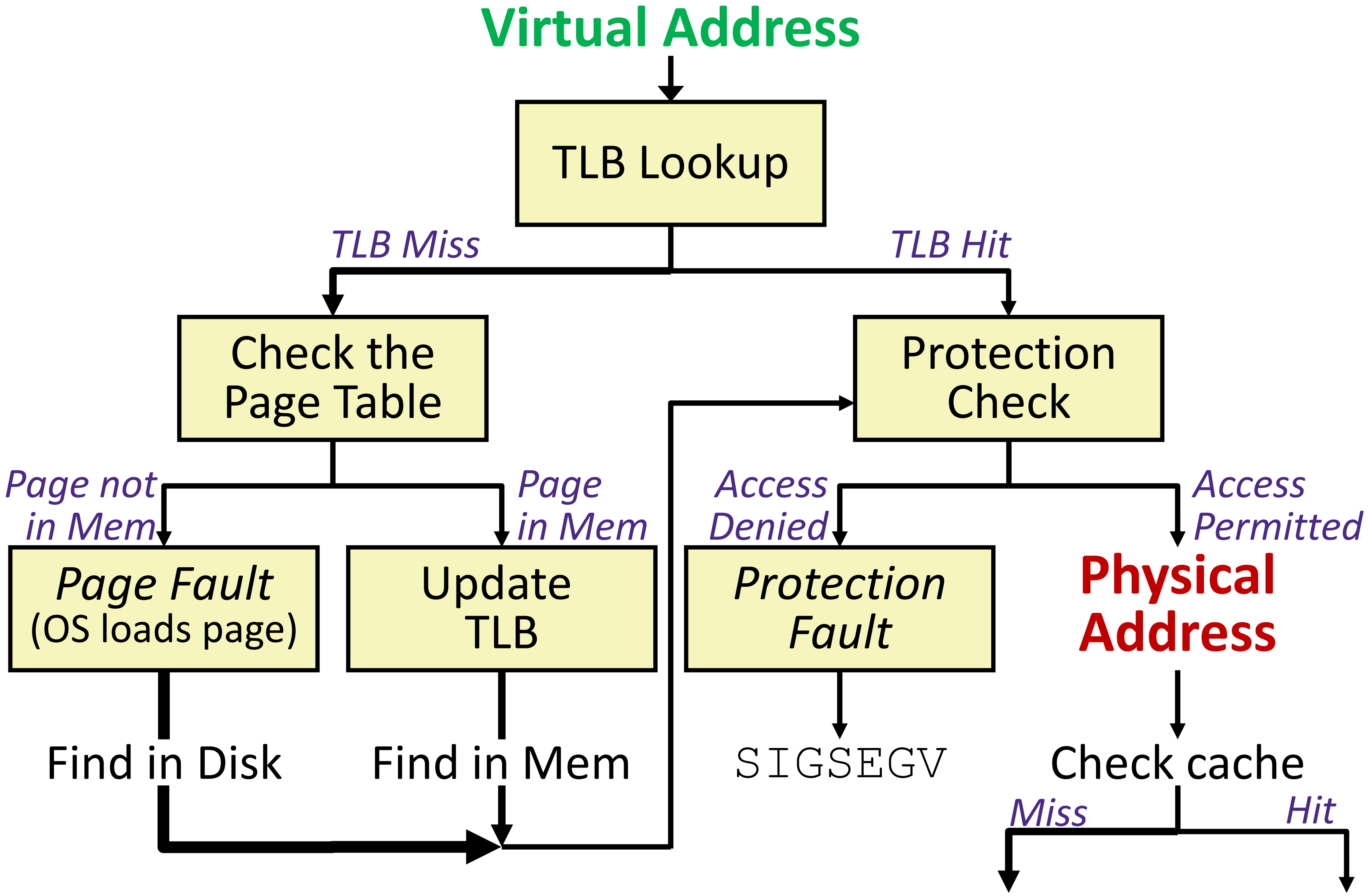


- A TLB miss incurs an additional memory access (the PTE)
    - Fortunately, TLB misses are rare

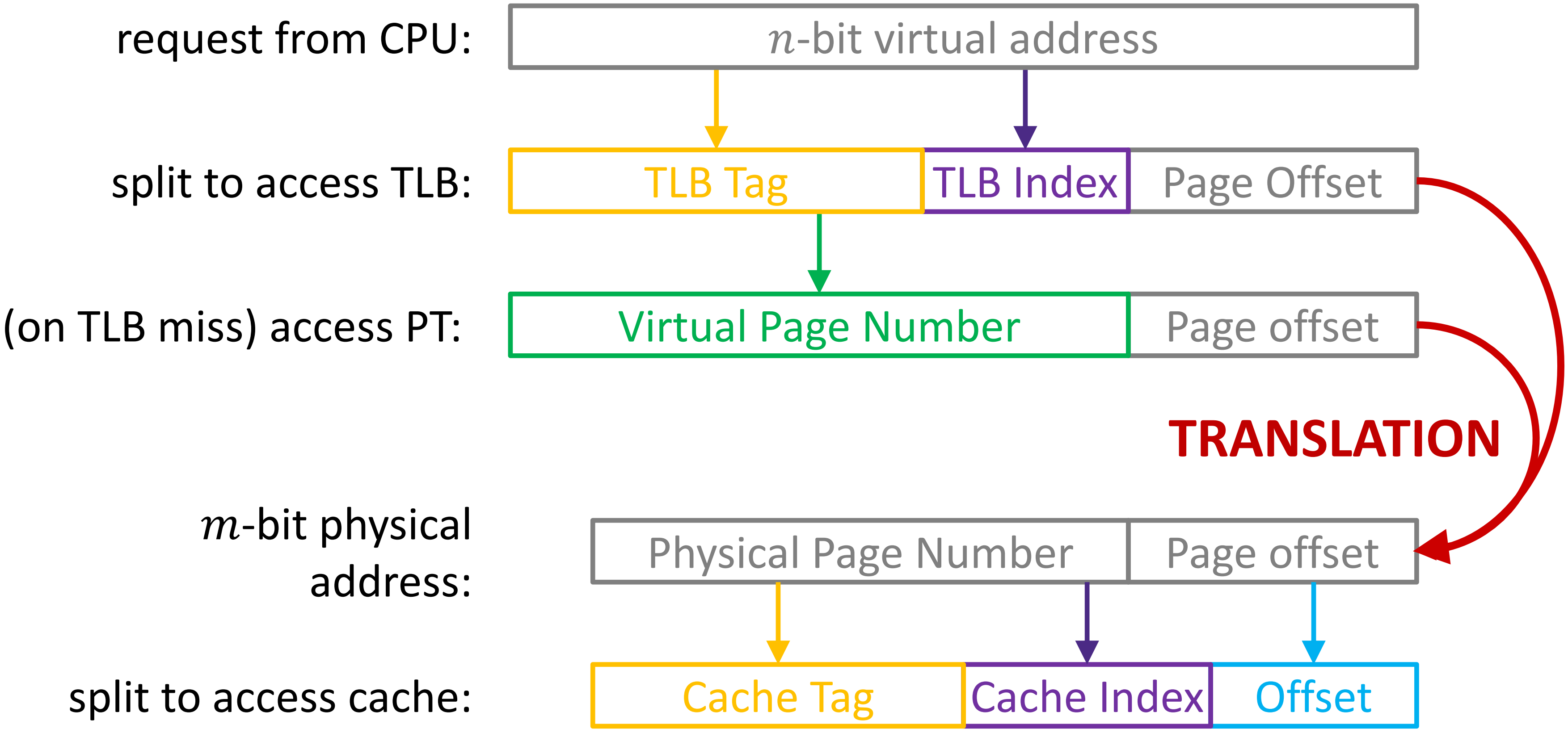
Fetching Data on a Memory Read

1) Check TLB
   - Input: VPN, Output: PPN
   - *TLB Hit:* Fetch translation, return PPN
   - *TLB Miss:* Check page table (in memory)
     - *Page Table Hit:* Load page table entry into TLB
     - *Page Fault:* Fetch page from disk to memory, update corresponding page table entry, then load entry into TLB
2) Check cache
   - Input: physical address, Output: data
   - *Cache Hit:* Return data value to processor
   - *Cache Miss:* Fetch data value from memory, store it in cache, return it to processor

Address Translation



Address Manipulation



request from CPU:

n-bit virtual address

split to access TLB:

TLB Tag | TLB Index | Page Offset

(on TLB miss) access PT:

Virtual Page Number | Page offset

TRANSLATION

m-bit physical address:

Physical Page Number | Page offset

split to access cache:

Cache Tag | Cache Index | Offset

Context Switching Revisited

- What needs to happen when the CPU switches processes?
  - Registers:
    - Save state of old process, load state of new process
    - Including the Page Table Base Register (PTBR)
  - Memory:
    - Nothing to do! Pages for processes already exist in memory/disk and protected from each other
  - TLB:
    - *Invalidate* all entries in TLB – mapping is for old process' VAs
  - Cache:
    - Can leave alone because storing based on PAs – good for shared data

Summary of Address Translation Symbols

- Basic Parameters

$\mathrm{N} = 2^n$	Number of addresses in virtual address space
$\mathrm{M} = 2^m$	Number of addresses in physical address space
$\mathrm{P} = 2^p$	Page size (bytes)

- Components of the virtual address (VA)

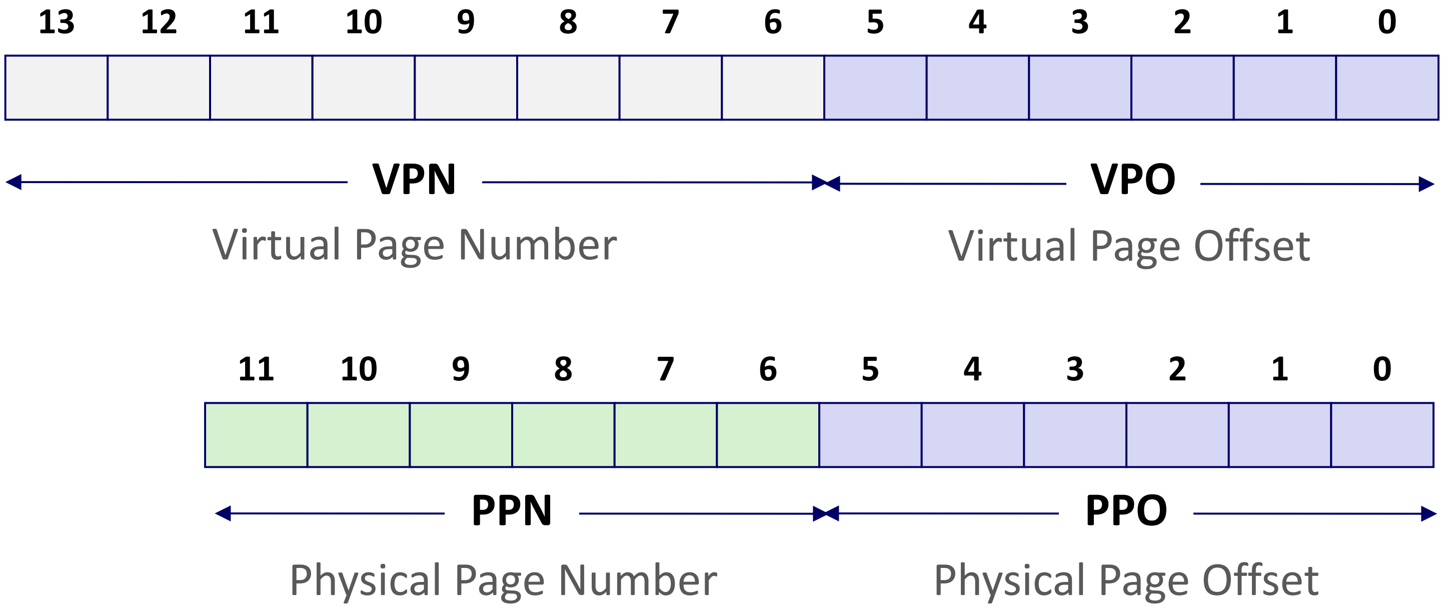
VPO	Virtual page offset
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag

- Components of the physical address (PA)

PPO	Physical page offset (same as VPO)
PPN	Physical page number

Simple Memory System Example (small)

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



Simple Memory System: Page Table

- Only showing first 16 entries (out of ______)
  - **Note:** showing 2 hex digits for PPN even though only 6 bits
  - **Note:** other management bits not shown, but part of PTE

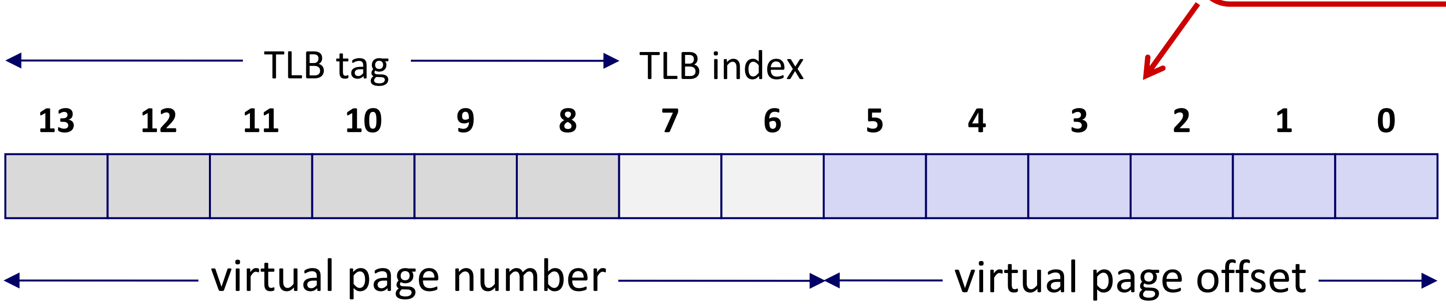
VPN	PPN	Valid
0	28	1
1	–	0
2	33	1
3	02	1
4	–	0
5	16	1
6	–	0
7	–	0

VPN	PPN	Valid
8	13	1
9	17	1
A	09	1
B	–	0
C	–	0
D	2D	1
E	–	0
F	0D	1

Simple Memory System: TLB

- 16 entries total
- 4-way set associative

Why does the TLB ignore the page offset?

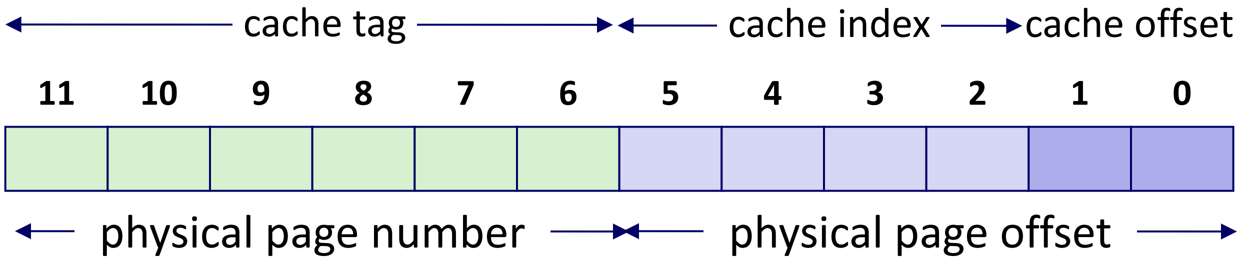


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System: Cache

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Direct-mapped with K = 4 B, C/K = 16
- Physically addressed



cache tag | cache index | cache offset

11 10 9 8 7 6 5 4 3 2 1 0

physical page number | physical page offset

Index	Tag	Valid	B0	B1	B2	B3
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

Index	Tag	Valid	B0	B1	B2	B3
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	–	–	–	–

Current State of Memory System

TLB:

Set	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V	Tag	PPN	V
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

Page table (partial):

VPN	PPN	V	VPN	PPN	V
0	28	1	8	13	1
1	–	0	9	17	1
2	33	1	A	09	1
3	02	1	B	–	0
4	–	0	C	–	0
5	16	1	D	2D	1
6	–	0	E	–	0
7	–	0	F	0D	1

Cache:

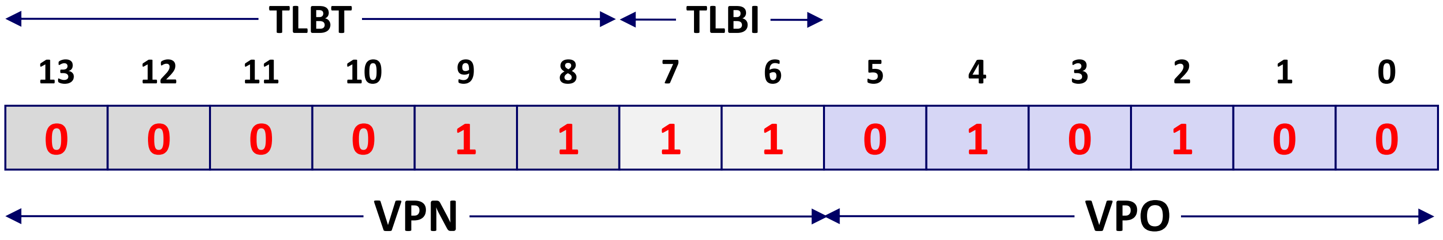
Index	Tag	V	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Index	Tag	V	B0	B1	B2	B3
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	–	–	–	–

Memory Request Example #1

Note: It is just coincidence that the PPN is the same width as the cache Tag

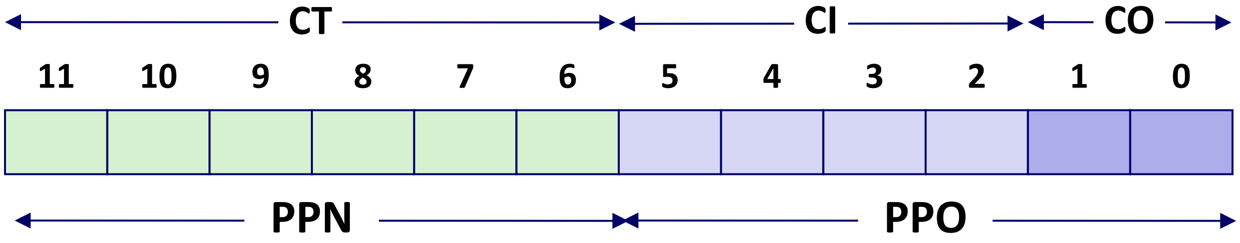
- Virtual Address: `0x03D4`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

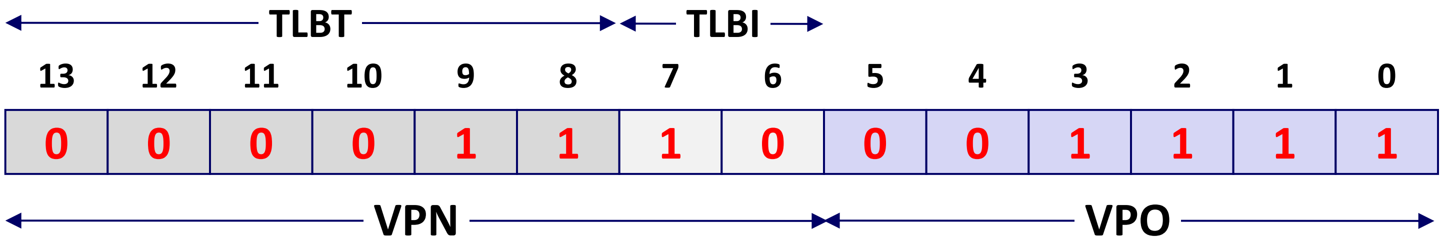


CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #2

Note: It is just coincidence that the PPN is the same width as the cache Tag

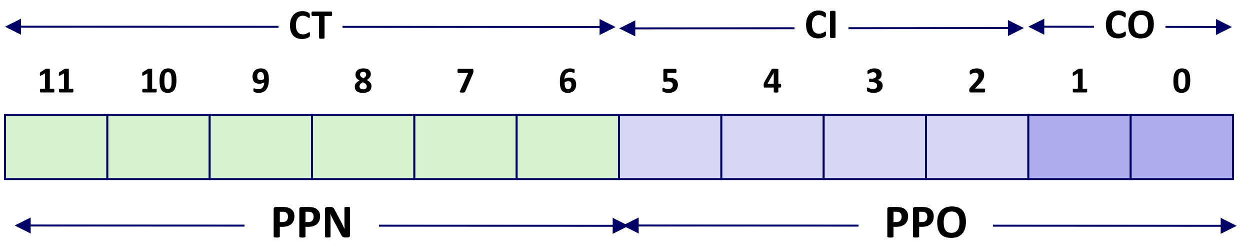
- Virtual Address: `0x038F`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	0	1	1	1	1

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:



11	10	9	8	7	6	5	4	3	2	1	0

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #3

Note: It is just coincidence that the PPN is the same width as the cache Tag

- Virtual Address: `0x0020`

TLBT						TLBI							
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0
VPN								VPO					

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

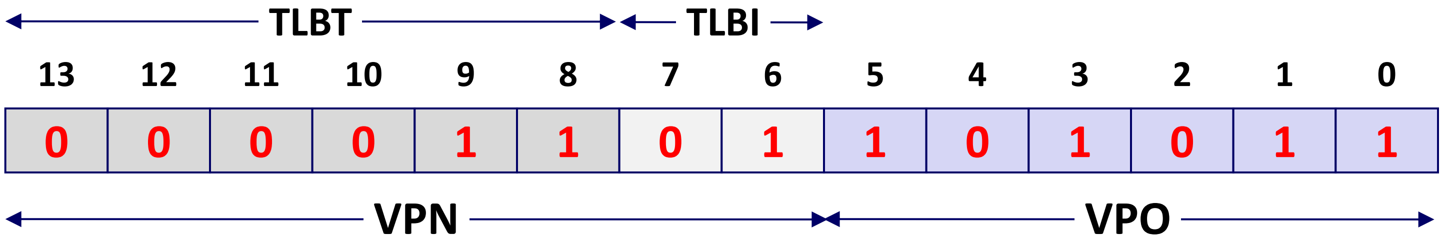
CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Request Example #4

Note: It is just coincidence that the PPN is the same width as the cache Tag

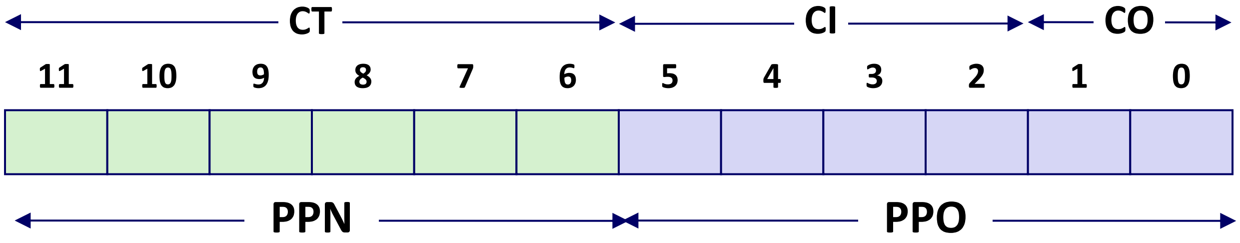
- Virtual Address: `0x036B`



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	0	1	1

VPN ______ TLBT _____ TLBI _____ TLB Hit? ___ Page Fault? ___ PPN _____

- Physical Address:

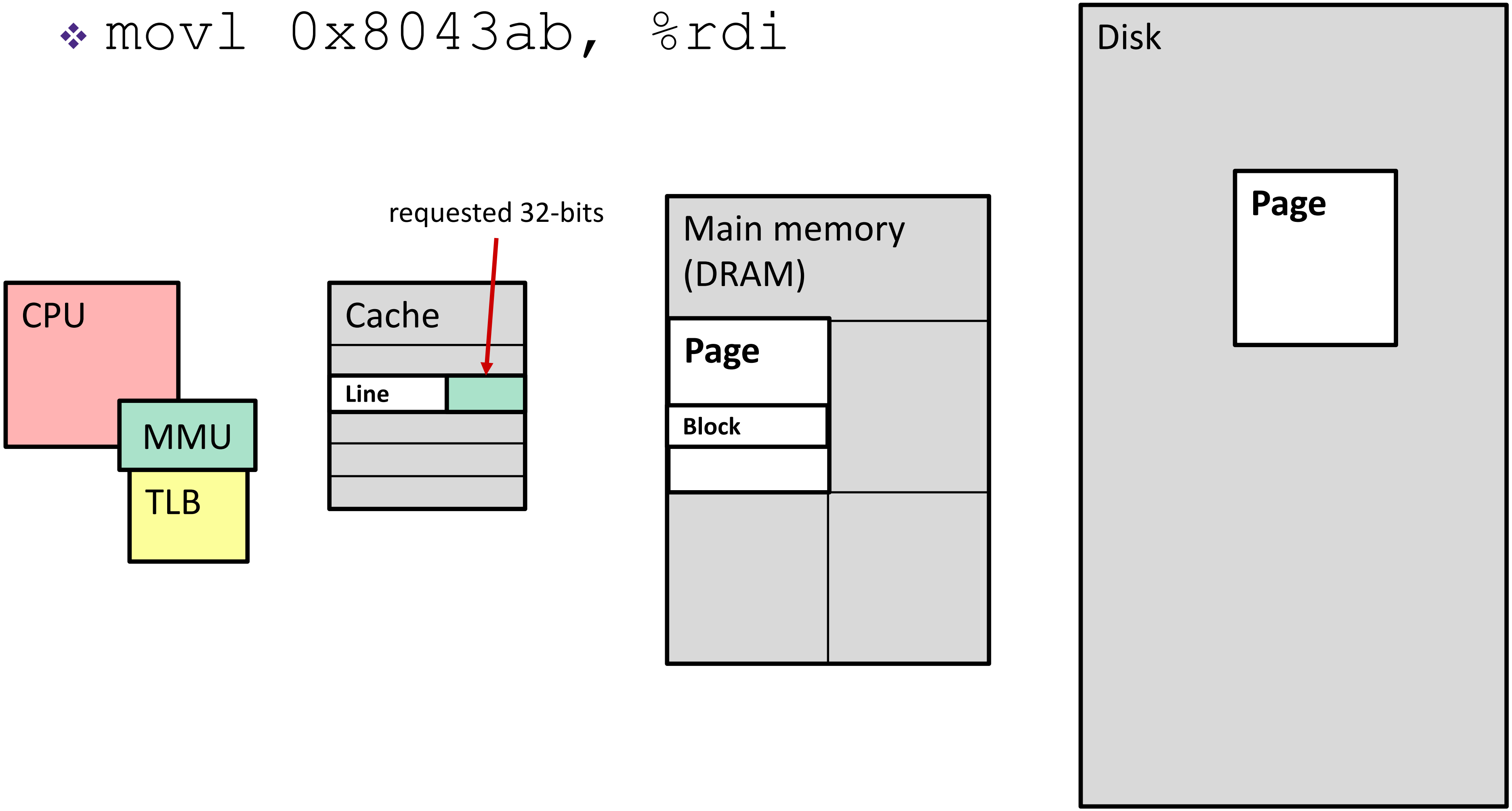


11	10	9	8	7	6	5	4	3	2	1	0

CT ______ CI _____ CO _____ Cache Hit? ___ Data (byte) _______

Memory Overview

- `movl 0x8043ab, %rdi`



Page Table Reality

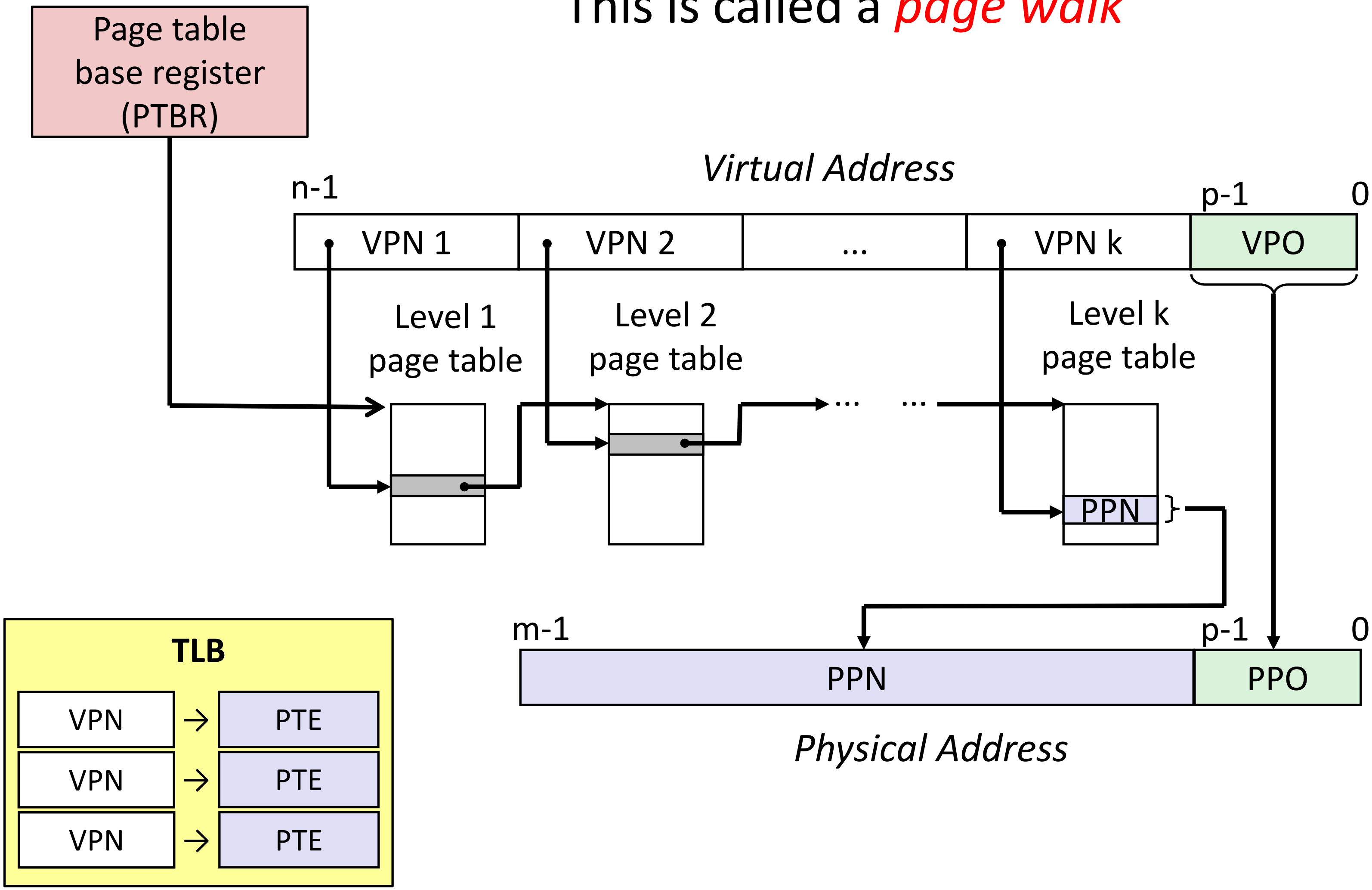
This is extra (non-testable) material

- Just one issue… the numbers don't work out for the story so far!

- The problem is the page table for each process:
  - Suppose 64-bit VAs, 8 KiB pages, 8 GiB physical memory
  - How many page table entries is that?
  - About how long is each PTE?
  - **Moral:** Cannot use this naïve implementation of the virtual→physical page mapping – it's *way* too big

A Solution: Multi-level Page Tables

This is extra (non-testable) material



This is called a *page walk*

Page table base register (PTBR)

Virtual Address

Physical Address

TLB

VPN → PTE

VPN → PTE

VPN → PTE

Multi-level Page Tables

This is extra
(non-testable)
material

- A tree of depth k where each node at depth i has up to 2^j children if part i of the VPN has j bits
- Hardware for multi-level page tables inherently more complicated
  - But it's a necessary complexity – 1-level does not fit
- Why it works: Most subtrees are not used at all, so they are never created and definitely aren't in physical memory
  - Parts created can be evicted from cache/memory when not being used
  - Each node can have a size of ~1-100KB
- But now for a k-level page table, a TLB miss requires $k + 1$ cache/memory accesses
  - Fine so long as TLB misses are rare – motivates larger TLBs

Practice VM Question

- Our system has the following properties
  - 1 MiB of physical address space
  - 4 GiB of virtual address space
  - 32 KiB page size
  - 4-entry fully associative TLB with LRU replacement

a) Fill in the following blanks:

_________ Entries in a page table

_________ Minimum bit-width of PTBR

_________ TLBT bits

_________ Max # of valid entries in a page table

Practice VM Question

- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

b) What is the largest stride (in bytes) between successive memory accesses (in the VA space)?

Practice VM Question

- One process uses a page-aligned *square* matrix `mat[]` of 32-bit integers in the code shown below:

```
#define MAT_SIZE = 2048
for(int i = 0; i < MAT_SIZE; i++)
  mat[i*(MAT_SIZE+1)] = i;
```

c) Assuming all of `mat[]` starts on disk, what are the following hit rates for the execution of the for-loop?

________ TLB Hit Rate ________ Page Table Hit Rate