

Lecture 23: Virtual Memory II

Philipp Koehn

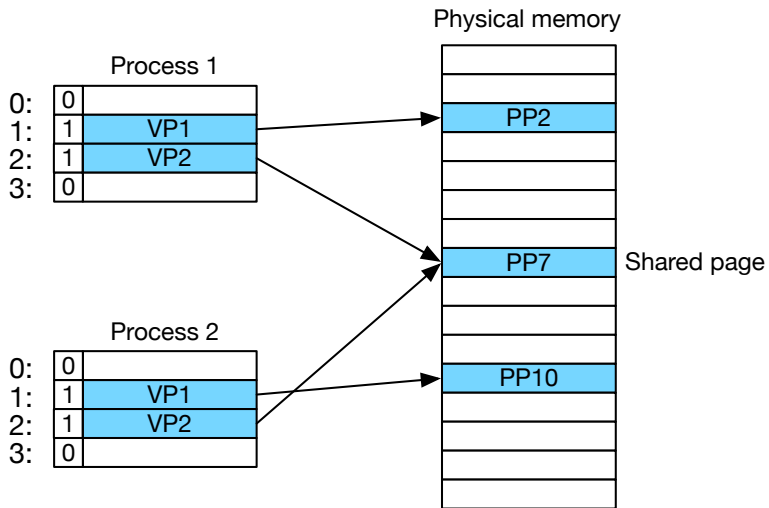
April 6, 2020

601.229 Computer Systems Fundamentals

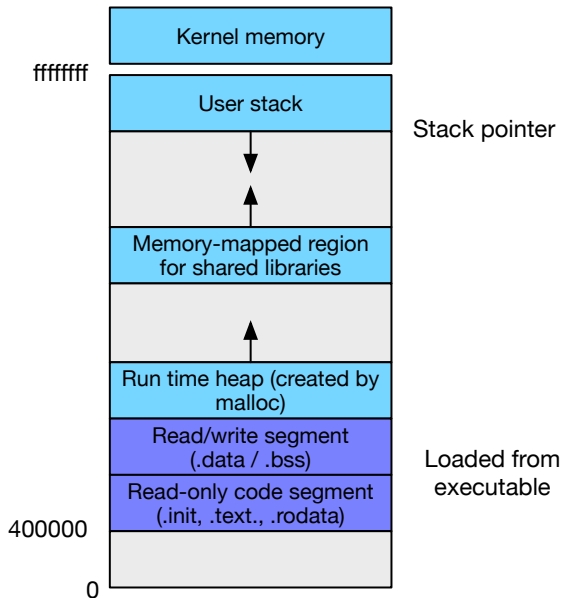


Memory management

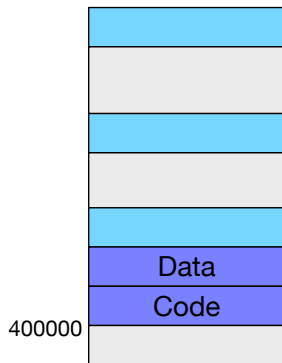
One Page Table per Process



Process Address Space



Simplified Linking



- ▶ Each process has its code in address `0x400000`
- ▶ Easy linking: Linker can establish fixed addresses

Simplified Loading

- ▶ When loading process into memory...
- ▶ Enter `.data` and `.text` section into page table

Simplified Loading

- ▶ When loading process into memory...
- ▶ Enter `.data` and `.text` section into page table
- ▶ Mark them as invalid (= not actually in RAM)

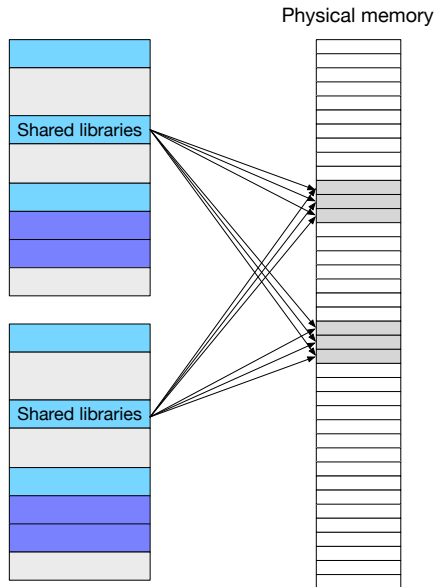
Simplified Loading

- ▶ When loading process into memory...
- ▶ Enter `.data` and `.text` section into page table
- ▶ Mark them as invalid (= not actually in RAM)
- ▶ Called memory mapping (more on that later)

Simplified Sharing

Shared libraries used by several processes: e.g., stdio providing printf, scanf, open, close, ...

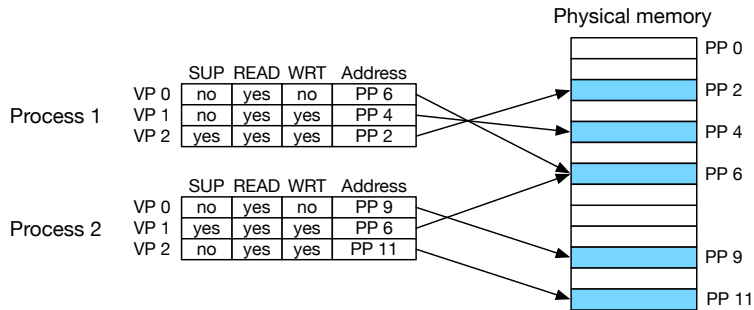
Not copied multiple times into RAM



Simplified Memory Allocation

- ▶ Process may need more memory (e.g., malloc call)
- ⇒ New entry in page table
- ▶ Mapped to arbitrary pages in physical memory
- ▶ Do not have to be contiguous

Memory Protection



- ▶ Page may be kernel only: SUP=yes
- ▶ Page may be read-only (e.g., code)

Address translation

Address Space

- ▶ Virtual memory size: $N = 2^n$ bytes
- ▶ Physical memory size: $M = 2^m$ bytes
- ▶ Page (block of memory): $P = 2^p$ bytes
- ▶ A virtual address can be encoded in n bits

Address Translation

- ▶ Task: mapping virtual address to physical address
 - ▶ virtual address (VA): used by machine code instructions
 - ▶ physical address (PA): location in RAM
- ▶ Formally

$$\text{MAP: } VA \rightarrow PA \cup 0$$

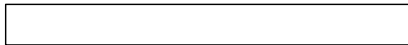
where:

$$\begin{aligned}\text{MAP}(A) &= PA \text{ if in RAM} \\ &= 0 \text{ otherwise}\end{aligned}$$

- ▶ Note: this happens very frequently in machine code
- ▶ We will do this in hardware: Memory Management Unit (MMU)

Basic Architecture

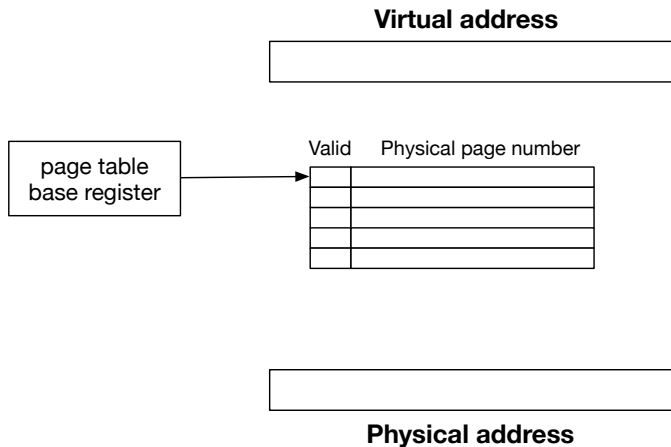
Virtual address



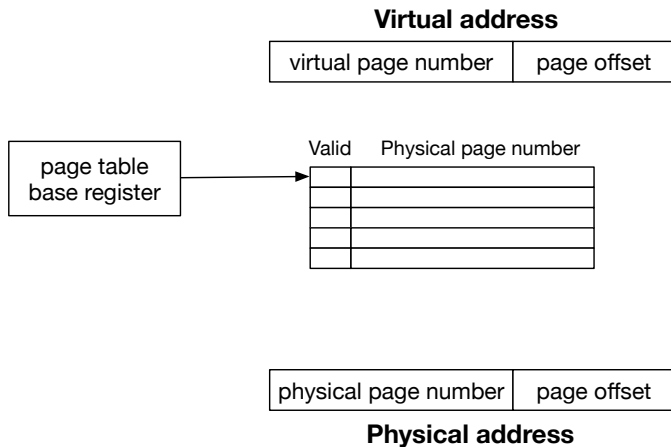
Physical address



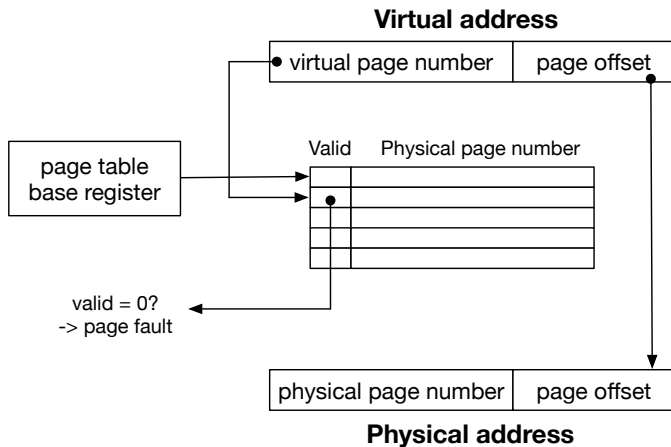
Basic Architecture



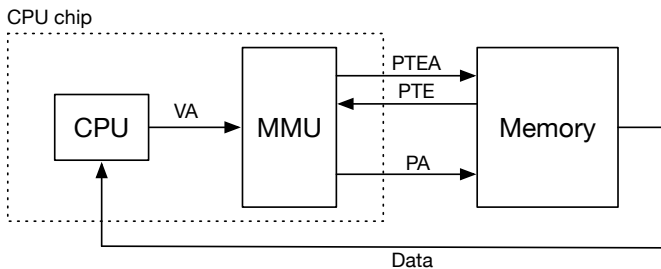
Basic Architecture



Basic Architecture

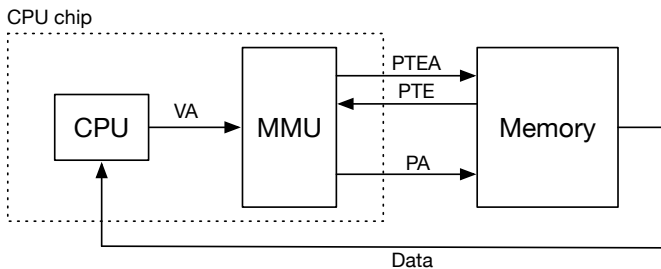


Page Hit



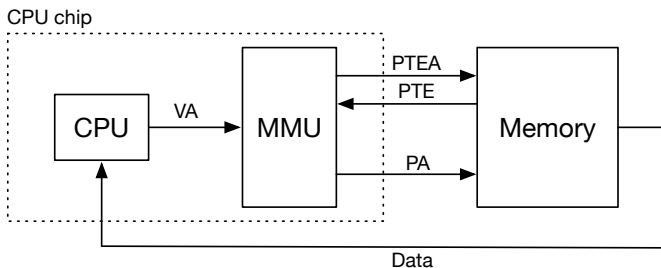
- ▶ VA: CPU requests data at virtual address

Page Hit



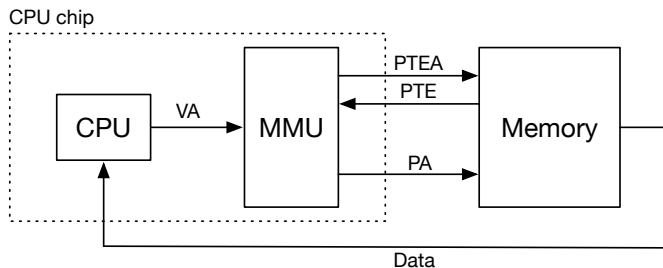
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table

Page Hit



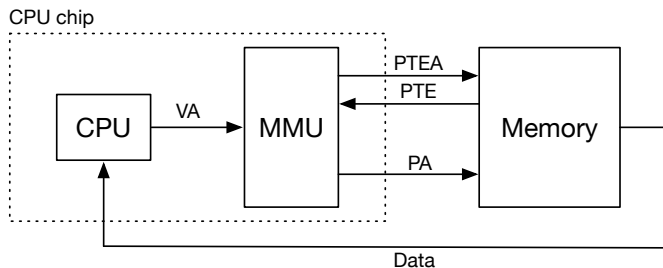
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry

Page Hit



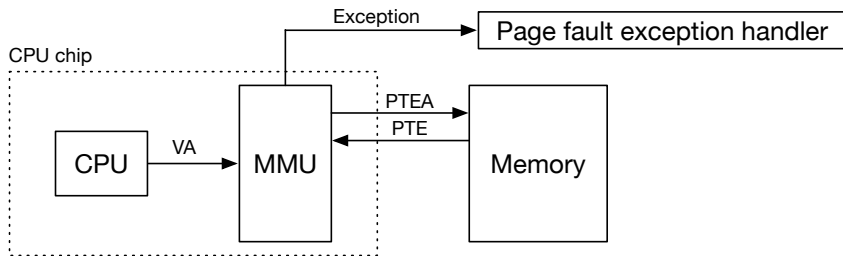
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry
- ▶ PA: get physical address from entry, look up in memory

Page Hit



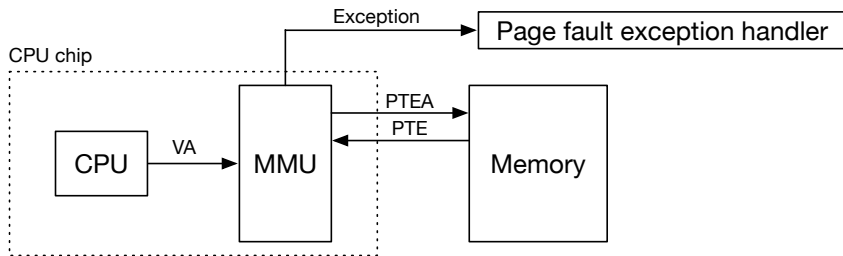
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry
- ▶ PA: get physical address from entry, look up in memory
- ▶ Data: returns data from memory to CPU

Page Fault



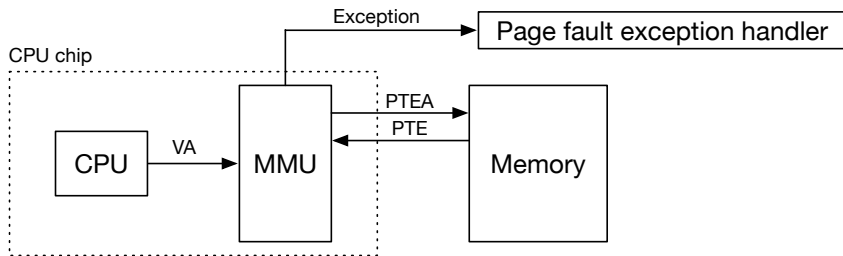
- ▶ VA: CPU requests data at virtual address

Page Fault



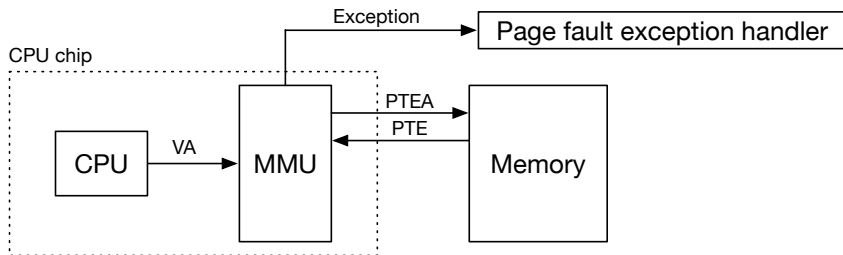
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table

Page Fault



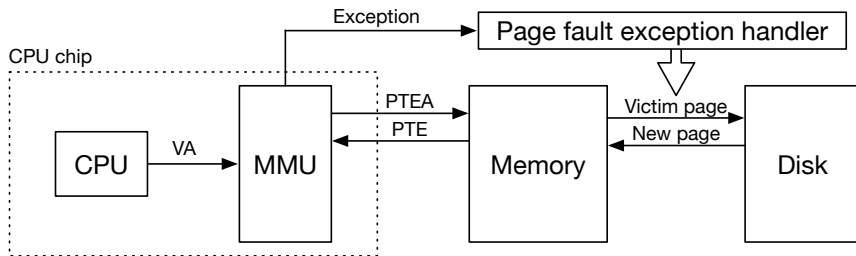
- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry

Page Fault



- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry
- ▶ Exception: page not in physical memory

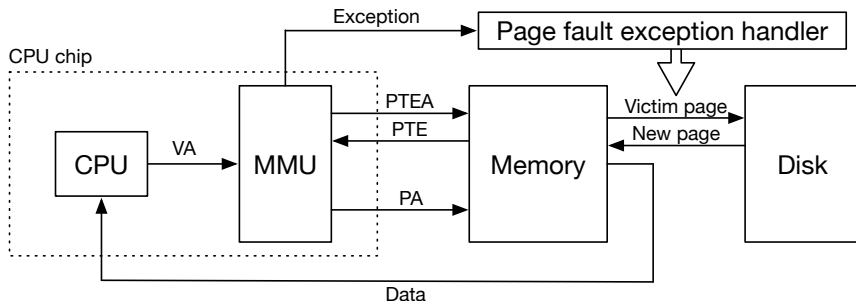
Page Fault



- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry
- ▶ Exception: page not in physical memory
- ▶ Page fault exception handler

- ▶ victim page to disk
- ▶ new page to memory
- ▶ update page table entries

Page Fault



- ▶ VA: CPU requests data at virtual address
- ▶ PTEA: look up page table entry in page table
- ▶ PTE: returns page table entry
- ▶ Exception: page not in physical memory
- ▶ Page fault exception handler

- ▶ victim page to disk
- ▶ new page to memory
- ▶ update page table entries
- ▶ **Re-do memory request**

Page Miss Exception

- ▶ Complex task
 - ▶ identify which page to remove from RAM (victim page)
 - ▶ load page from disk to RAM
 - ▶ update page table entry
 - ▶ trigger do-over of instruction that caused exception
- ▶ Note
 - ▶ loading into RAM very slow
 - ▶ added complexity of handling in software no big deal

Zoom poll!

Given the following code:

```
int arr[10000], i;  
for (i = 0; i<10000; i++) {  
    arr[i] = i;  
}
```

Assume that the page size is 4096 bytes, and that the base address of the array `a` is an exact multiple of 4096. If the access to `a[i]` does not cause a page fault when `i=0`, then what is the next value of `i` where a page fault might occur?

- A. 1
- B. 512
- C. 1024
- D. 4096
- E. None of the above

Refinements

Refinements

- ▶ On-CPU cache
- ▶ Slow look-up time
- ▶ Huge address space
- ▶ Putting it all together

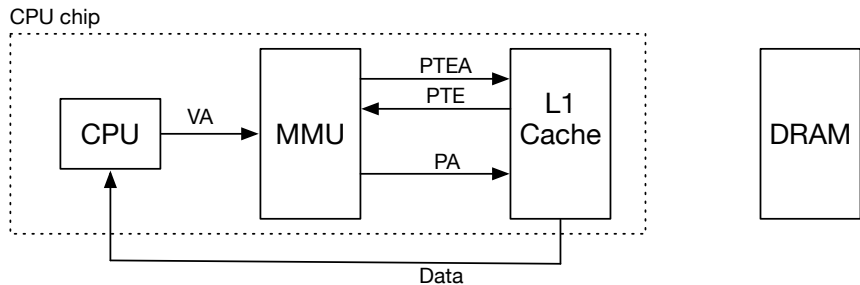
Refinements

- ▶ **On-CPU cache**
→ **integrate cache and virtual memory**
- ▶ Slow look-up time
- ▶ Huge address space
- ▶ Putting it all together

Integrating Caches and Virtual Memory

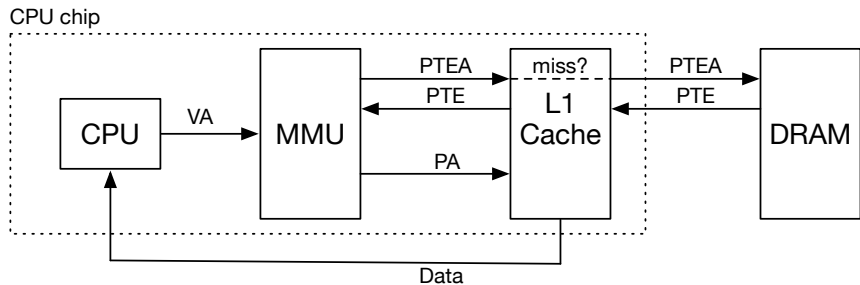
- ▶ Note
 - ▶ we claim that using on-disk memory is too slow
 - ▶ having data in RAM only practical solution
- ▶ Recall
 - ▶ we previously claimed that using RAM is too slow
 - ▶ having data in cache only practical solution
- ▶ Both true, so we need to combine

Integrating Caches and Virtual Memory



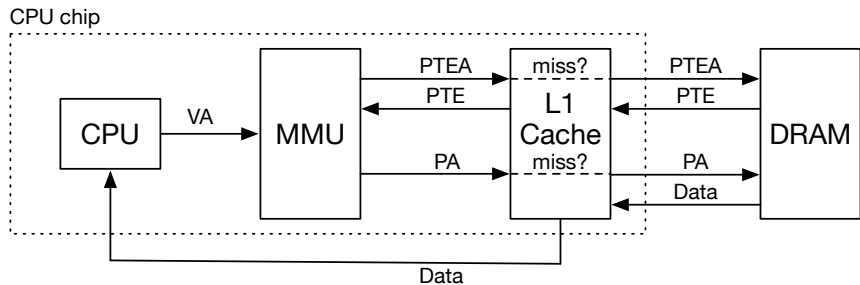
- ▶ MMU resolves virtual address to physical address
- ▶ Physical address is checked against cache

Integrating Caches and Virtual Memory



- ▶ Cache miss in page table retrieval?
- ⇒ Get page table from memory

Integrating Caches and Virtual Memory



- ▶ Cache miss in data retrieval?
- ⇒ Get data from memory

Refinements

- ▶ On-CPU cache
→ integrate cache and virtual memory
- ▶ **Slow look-up time**
→ **use translation lookahead buffer (TLB)**
- ▶ Huge address space
- ▶ Putting it all together

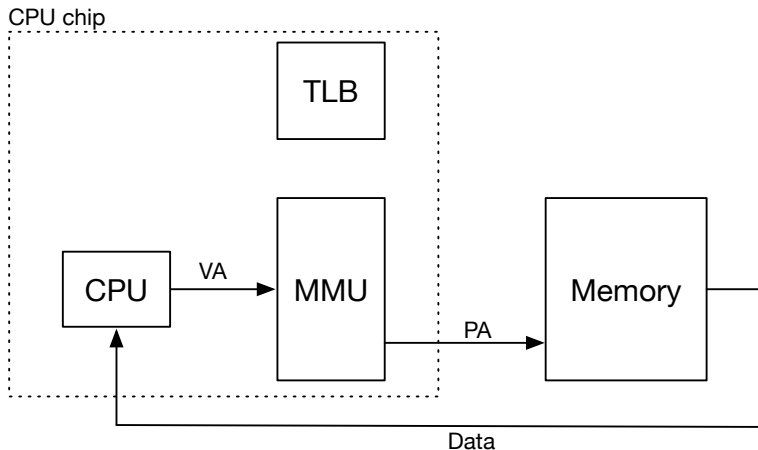
Look-Ups

- ▶ Every memory-related instruction must pass through MMU (virtual memory look-up)
- ▶ Very frequent, this has to be very fast
- ▶ Locality to the rescue
 - ▶ subsequent look-ups in same area of memory
 - ▶ look-up for a page can be cached

Translation Lookup Buffer

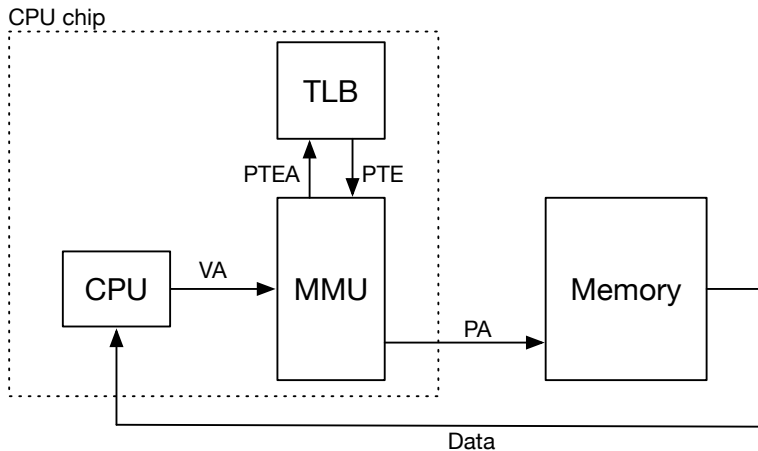
- ▶ Same structure as cache
- ▶ Break up address into 3 parts
 - ▶ lowest bits: offset in page
 - ▶ middle bits: index (location) in cache
 - ▶ highest bits: tag in cache
- ▶ Associative cache: more than one entry per index

Architecture



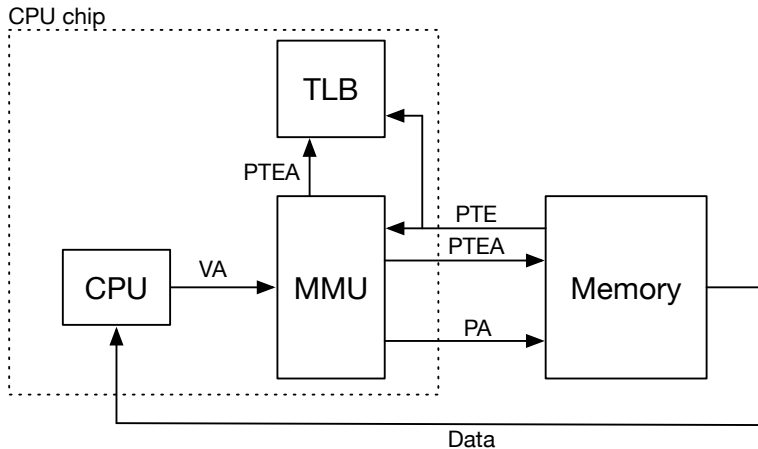
- ▶ Translation lookup buffer (TLB) on CPU chip

Translation Lookup Buffer (TLB) Hit



- ▶ Look up page table entry in TLB

Translation Lookup Buffer (TLB) Miss



- ▶ Page table entry not in TLB
- ▶ Retrieve page table entry from RAM