CSCI-UA.0201

Computer Systems Organization

Memory Management – Virtual Memory

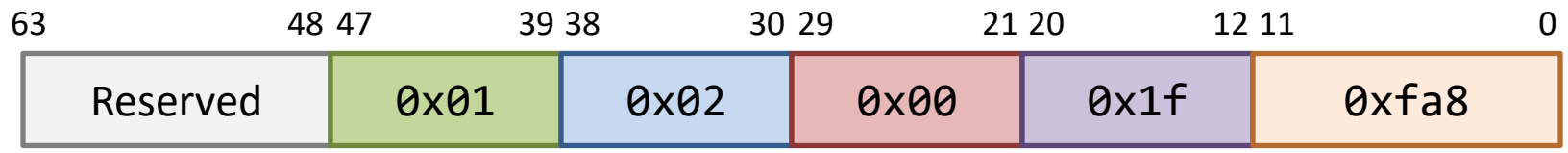
Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

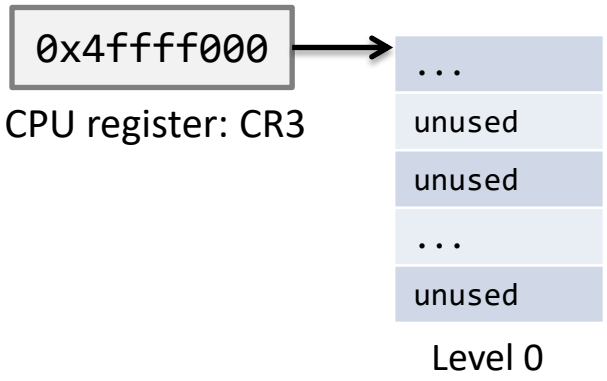
Demand Paging

- Memory Allocation (e.g., `p = sbrk(8192)`)
- User program to OS:
  - Declare a virtual address range from p to `p + 8192` for use by the current process.
- OS' actions:
  - Allocate the physical page and populate the page table.

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```



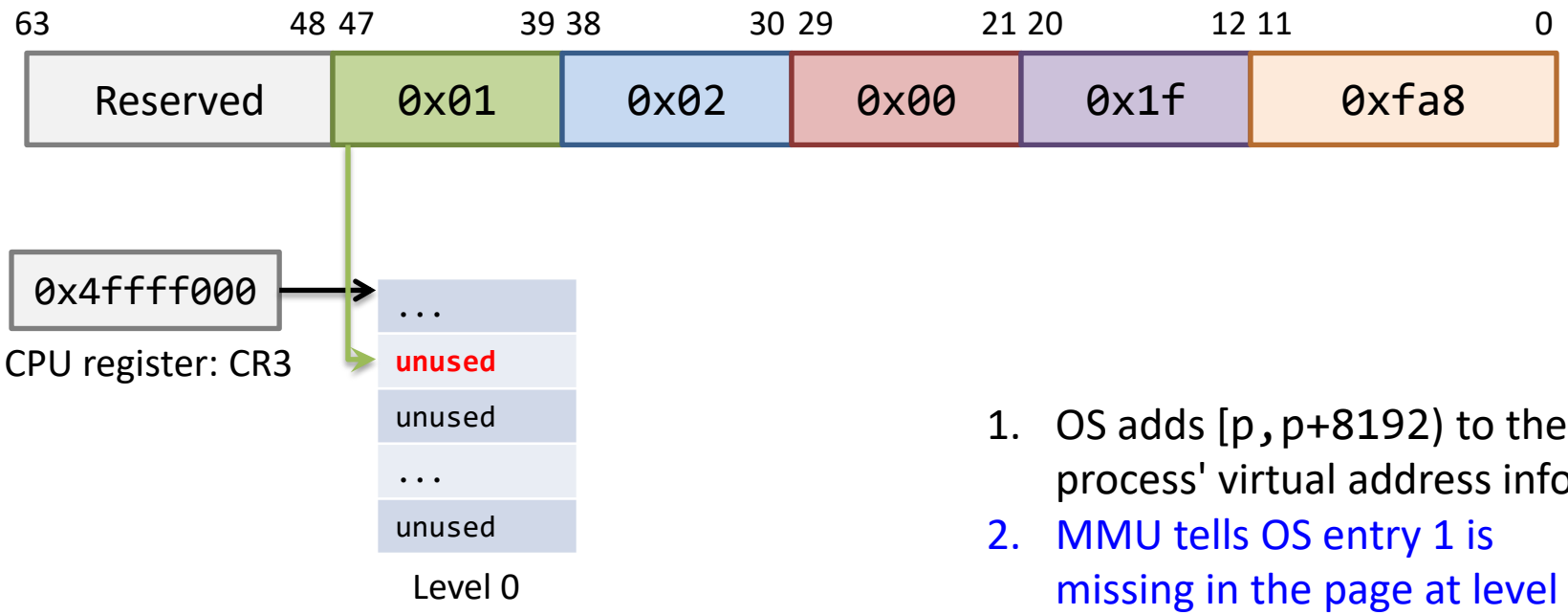


current process' page table

1. OS adds [p, p+8192) to the process' virtual address info

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

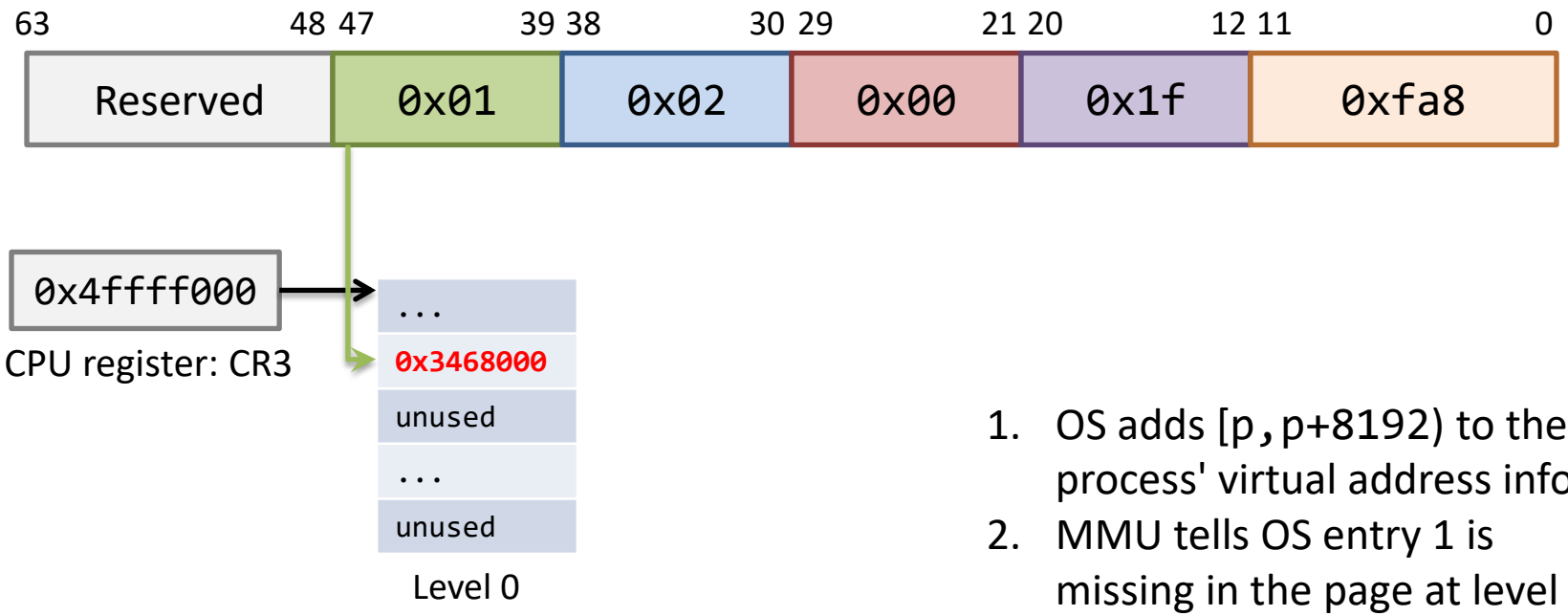


current process' page table

1. OS adds [p, p+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

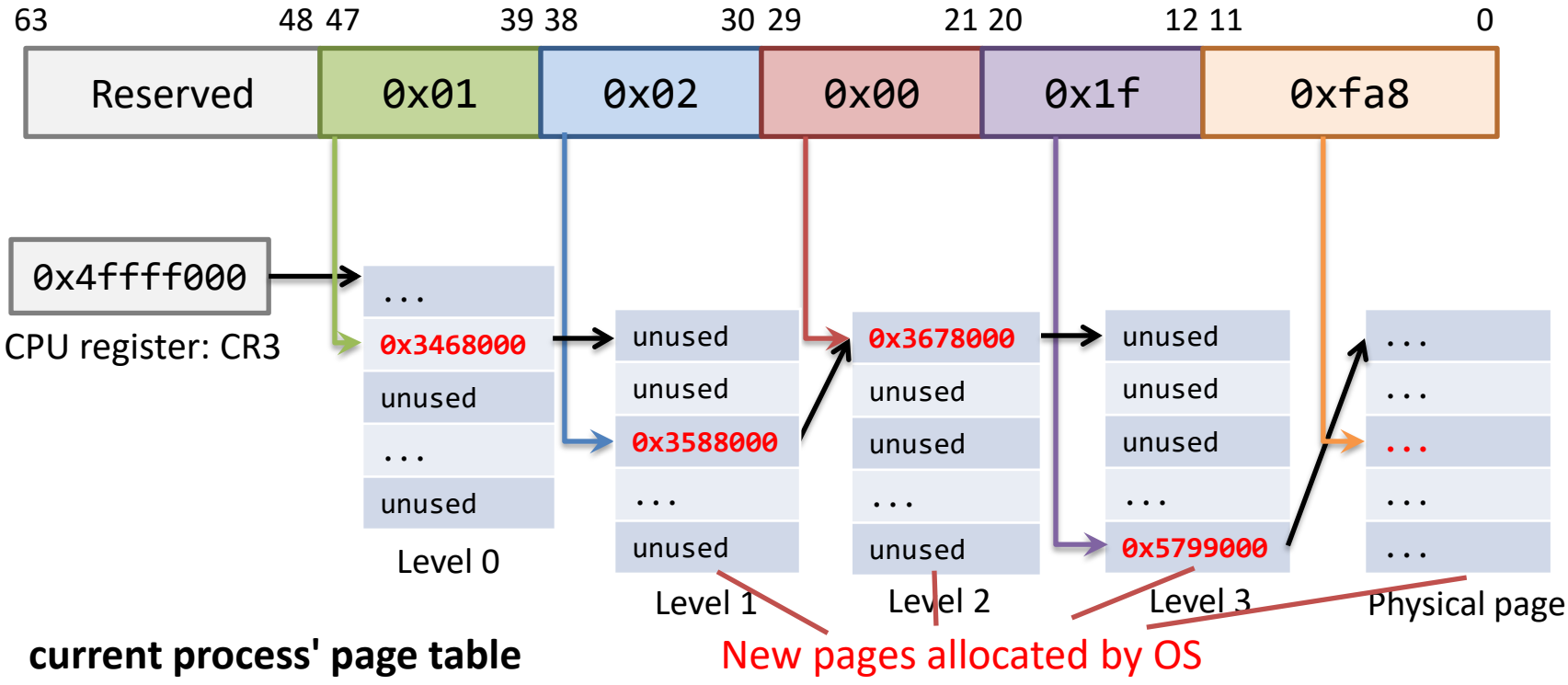


current process' page table

1. OS adds [p, p+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)
3. OS constructs the mapping for the address. (*Page fault handler*)

Demand Paging

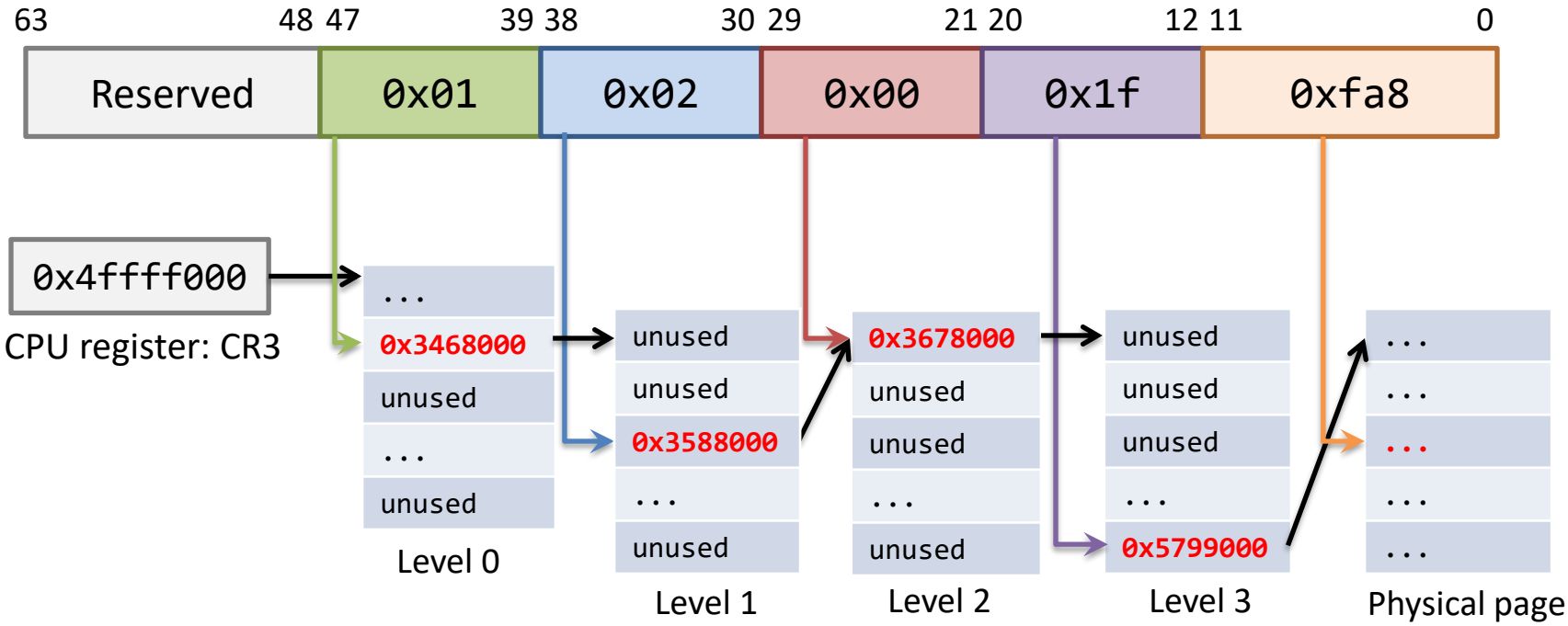
```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```



3. OS constructs the mapping for the address. (*Page fault handler*)

Demand Paging

```
    char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
→   p[0] = 'c'
    p[4096] = 's'
```

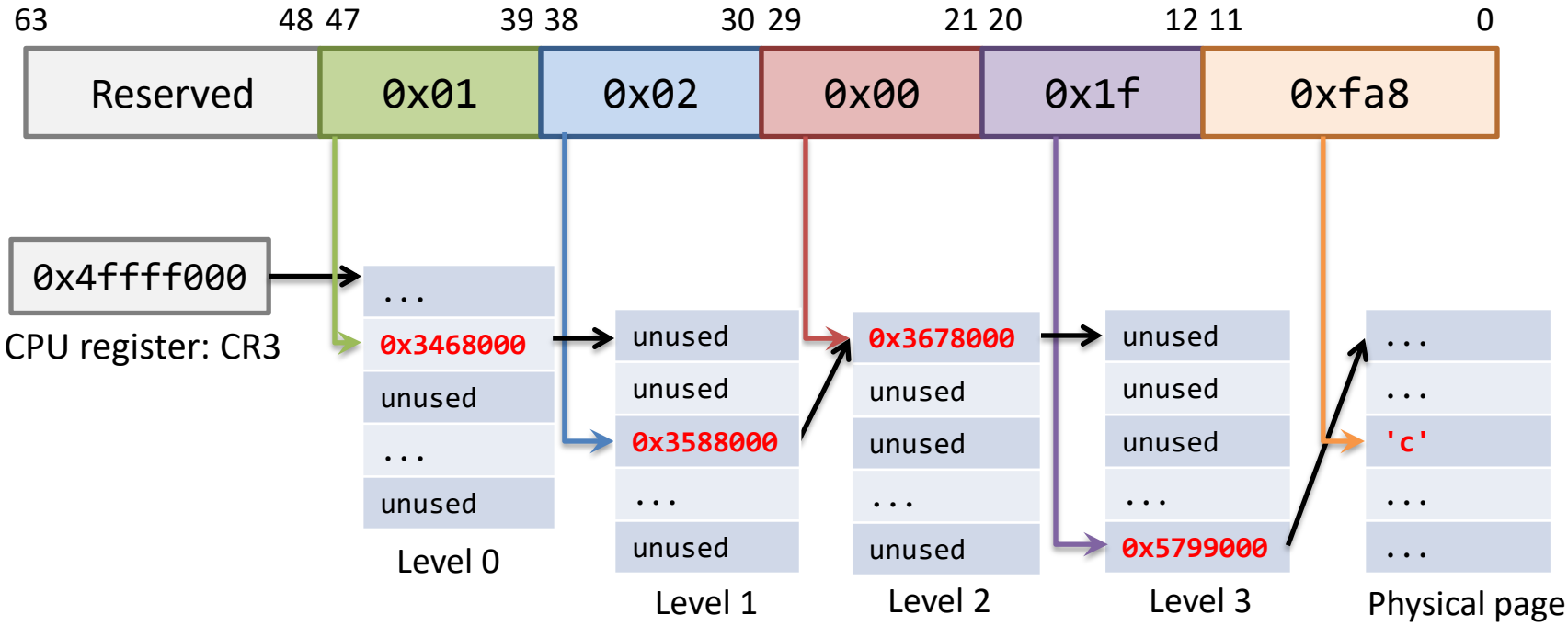


current process' page table

4. OS tells the CPU to resume execution

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

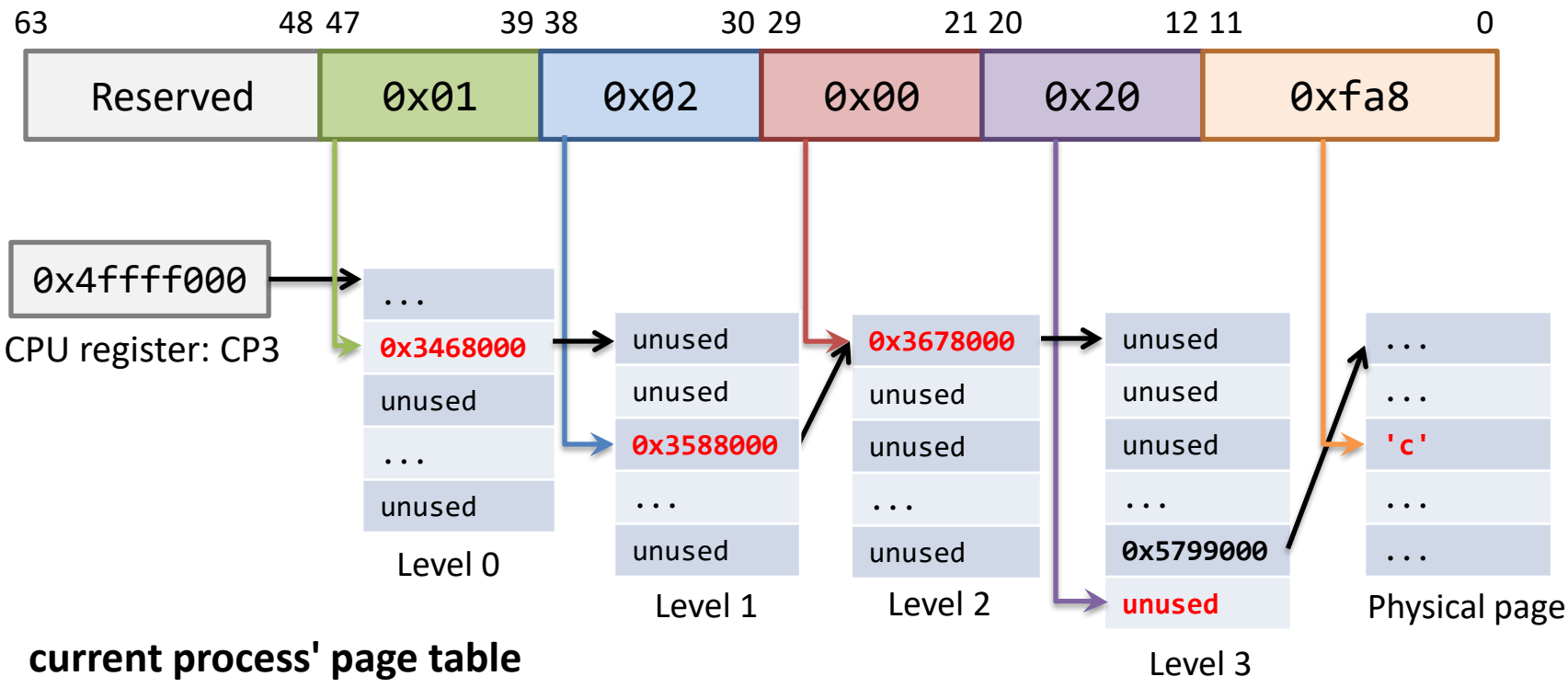


current process' page table

5. MMU translates address again and accesses the physical memory.

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'   // p+4096 is 0x01020020fa8
```



2. MMU tells OS entry 1 is missing in the page at level 3. (*Page fault*)

Understanding Segmentation Fault

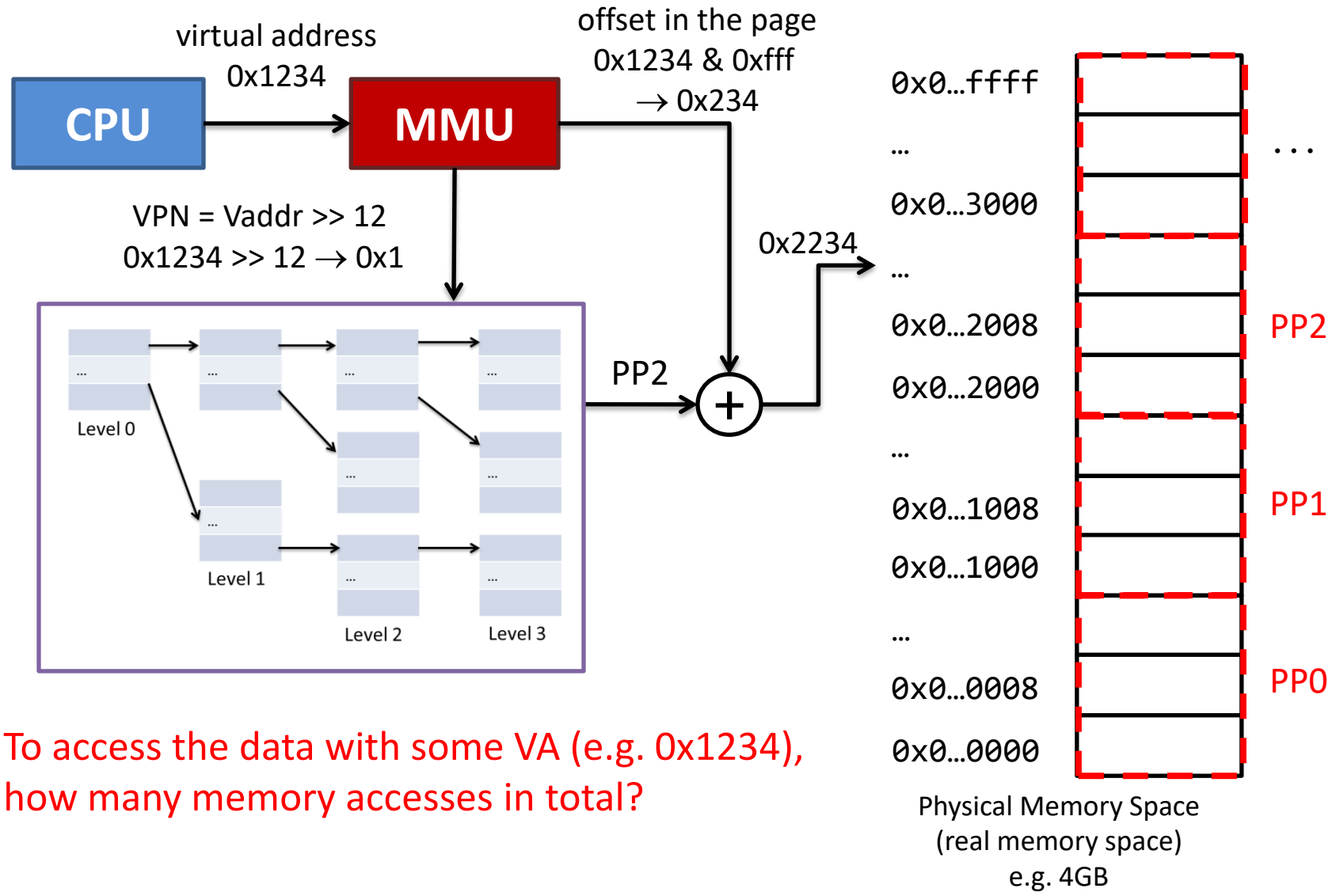
- Where does **segmentation fault** come from?
- Address translation can fail due to 2 reasons
  - MMU reads a missing page table entry (PTE)
    - PTE's present bit is unset
  - MMU reads a PTE with wrong permission for the access
    - write bit is unset for a write access
    - OS bit is set for user program access
- MMU generates "page fault", to be handled by OS
- OS either fixes the problem (e.g. demand paging) or aborts process with "segmentation fault"

Memory Access Cost

- Memory access latency
  - 100 ns
  - 160 ~ 200 CPU cycles

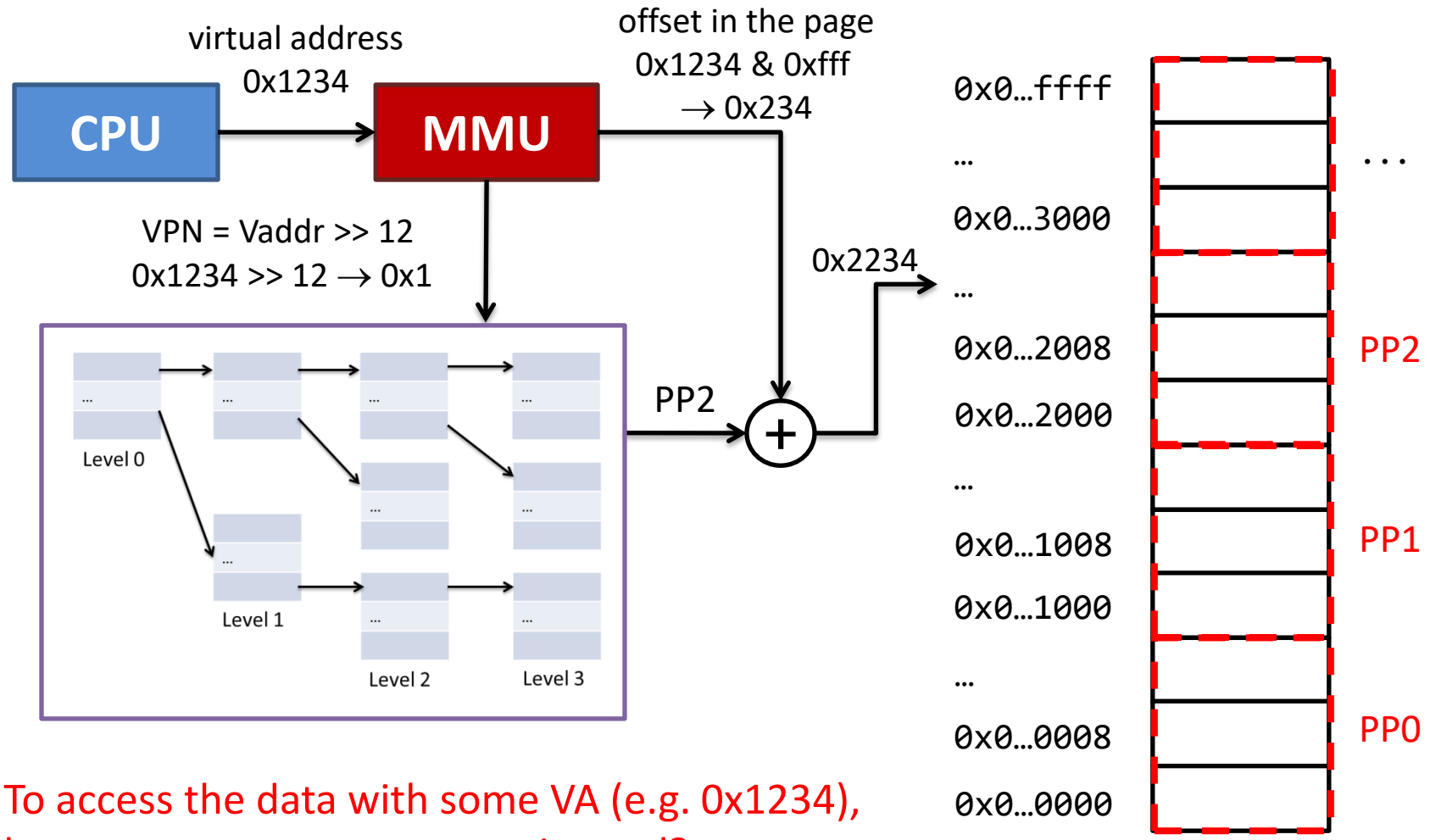
- Instructions that do not involve memory access can execute very quickly:
  - Instructions per CPU cycle >= 1

Address translation is potentially very costly



To access the data with some VA (e.g. 0x1234), how many memory accesses in total?

Address translation is potentially very costly

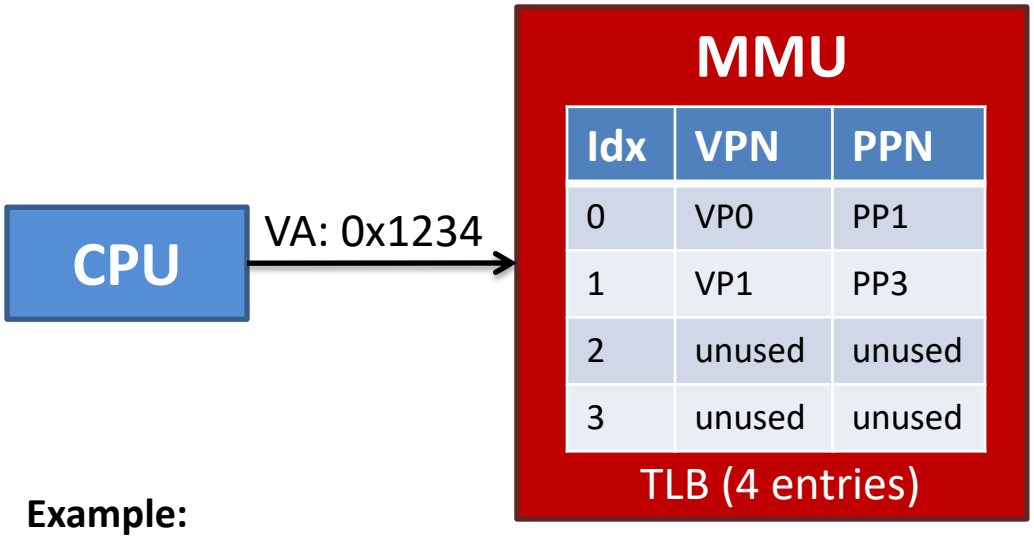


To access the data with some VA (e.g. 0x1234), how many memory accesses in total?

4 page table accesses plus one time data access which is **5** memory accesses

Speedup Address Translation

- Translation lookaside buffer (TLB)
  - Small cache in MMU
  - Maps virtual page numbers to physical page numbers

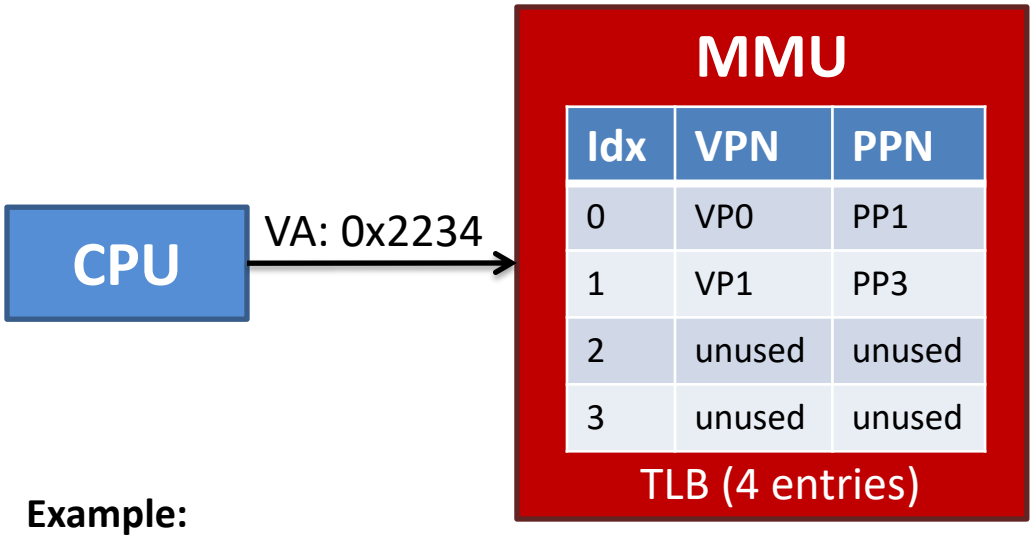


1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

Example:
VPN = 0x1234 >> 12 = 0x1
Index = 0x2 % 4 = 1
Check if TLB[1].VPN is VPN
Yes (TLB hit):
PA = PP3 + 0x234 = 0x3234

Speedup Address Translation

- Translation lookaside buffer (TLB)
  - Small cache in MMU
  - Maps virtual page numbers to physical page numbers

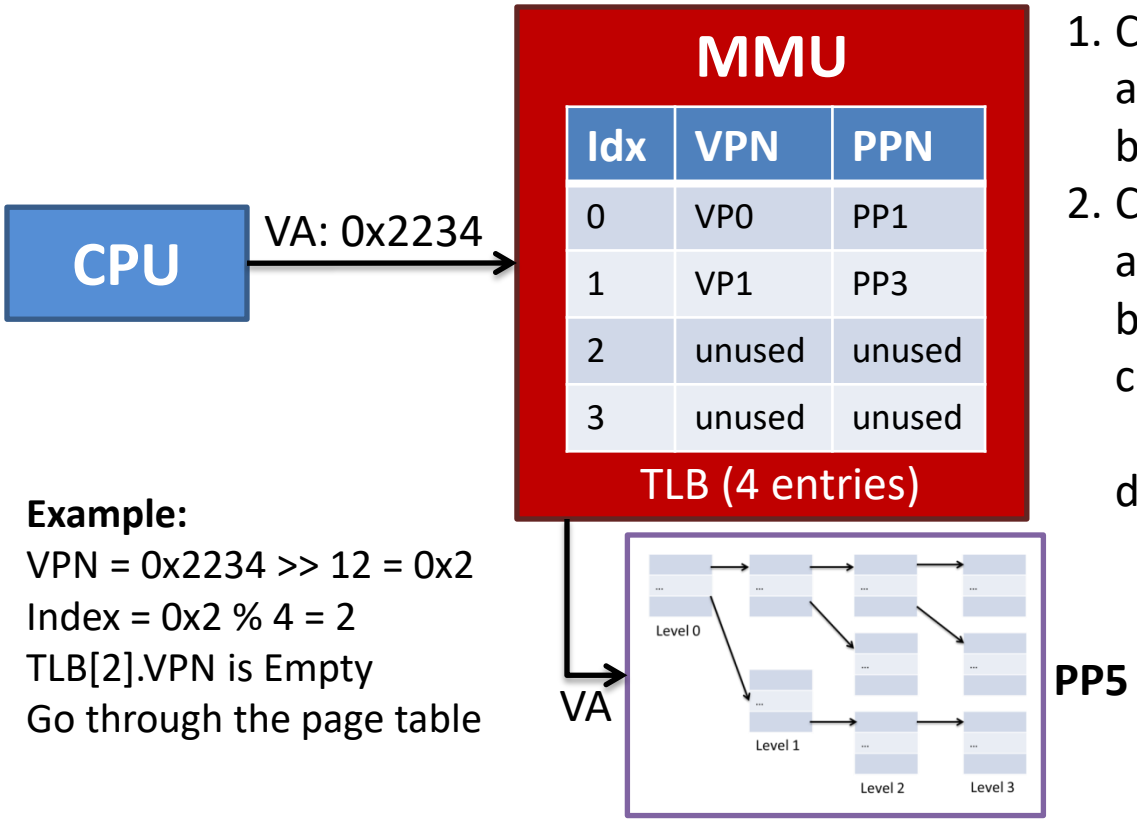


1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

Example:
VPN = 0x2234 >> 12 = 0x2
Index = 0x2 % 4 = 2
TLB[2].VPN is Empty

Speedup Address Translation

- Translation lookaside buffer (TLB)
  - Small cache in MMU
  - Maps virtual page numbers to physical page numbers

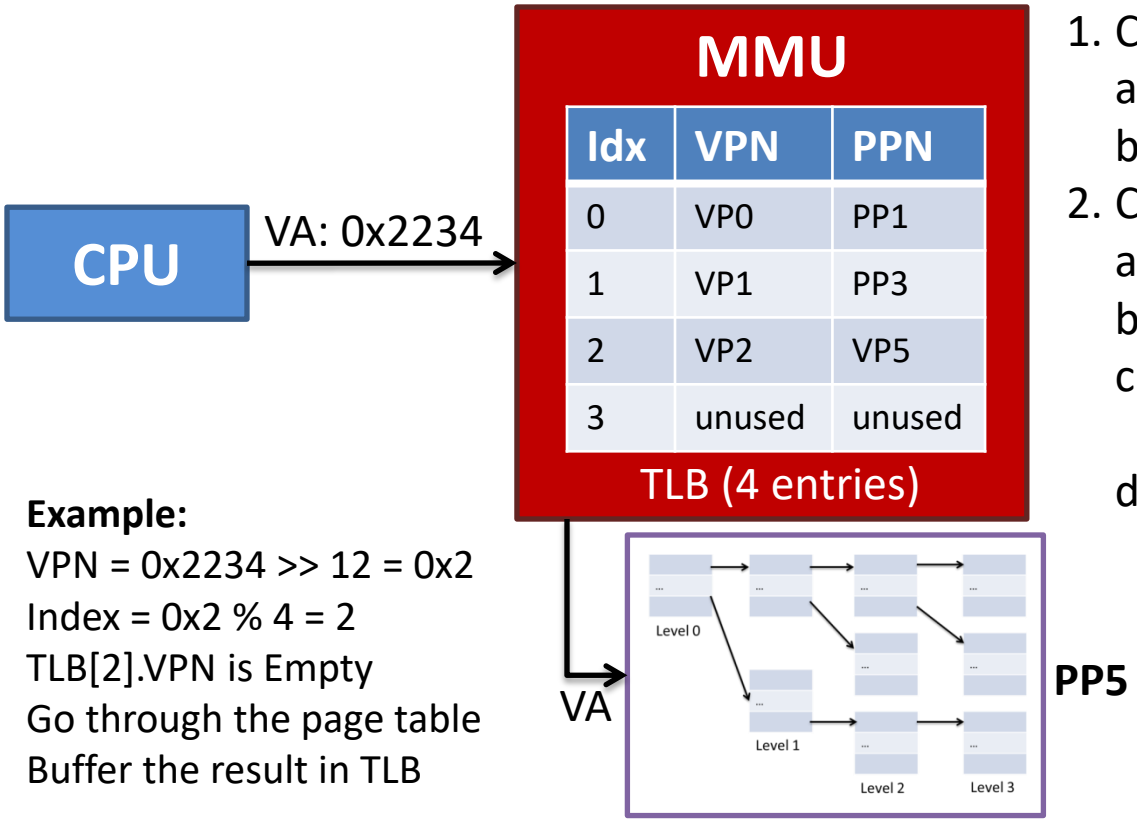


1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

Example:
VPN = 0x2234 >> 12 = 0x2
Index = 0x2 % 4 = 2
TLB[2].VPN is Empty
Go through the page table

Speedup Address Translation

- Translation lookaside buffer (TLB)
  - Small cache in MMU
  - Maps virtual page numbers to physical page numbers



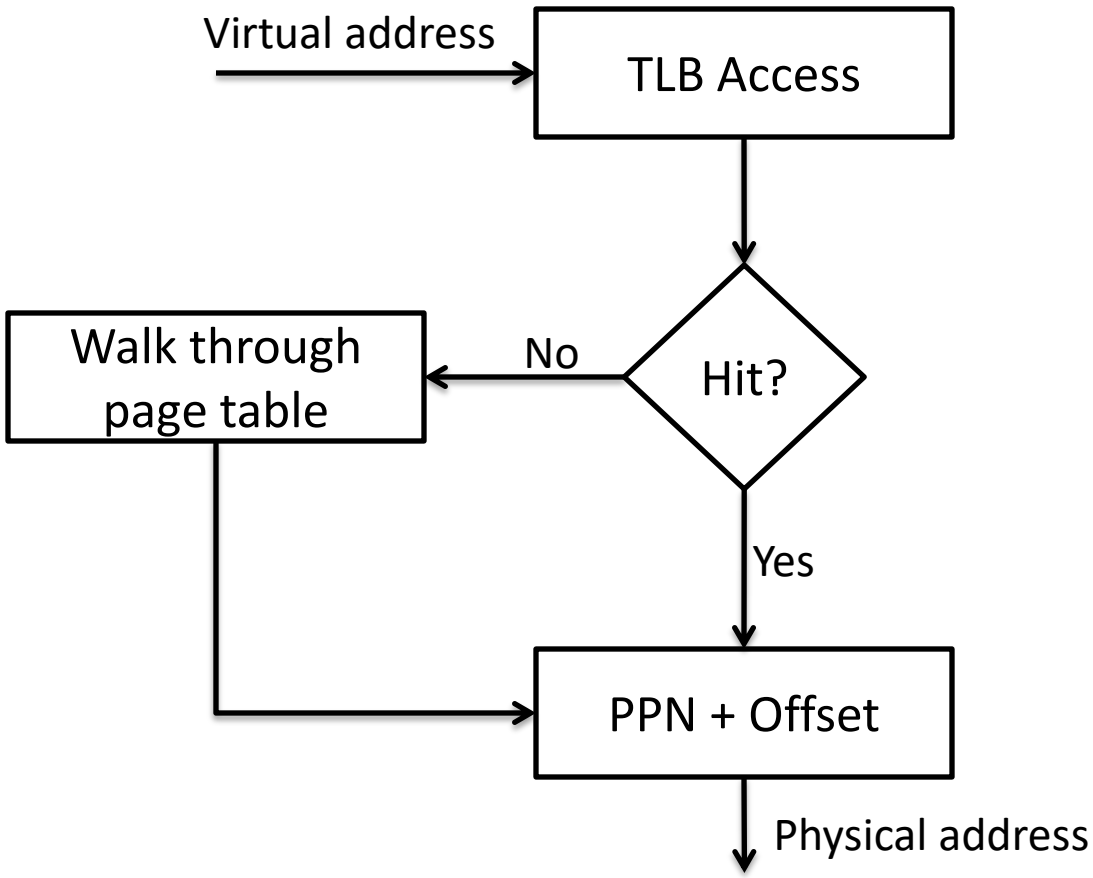
1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

Example:
VPN = 0x2234 >> 12 = 0x2
Index = 0x2 % 4 = 2
TLB[2].VPN is Empty
Go through the page table
Buffer the result in TLB

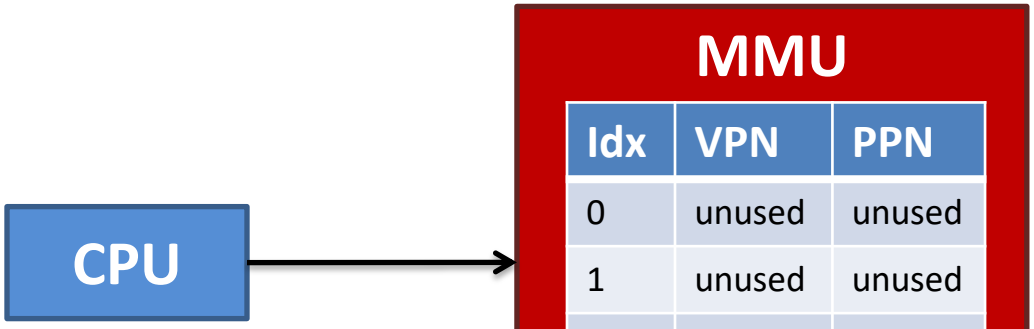
Latency

- Memory access
  - Hundreds of CPU cycles
- TLB access
  - Only a couple of CPU cycles

Summary


Virtual address
TLB Access
Hit?
No
Walk through page table
Yes
PPN + Offset
Physical address

Speedup Address Translation



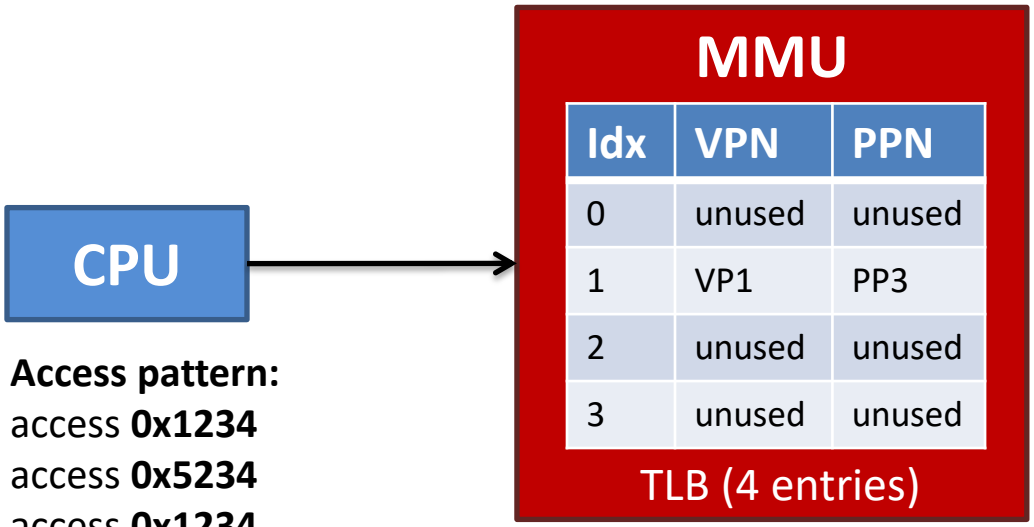
Access pattern:
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

TLB:
access **0x1234**, TLB Miss

1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

Both 0x1234 and 0x5234 go to entry 1

Speedup Address Translation



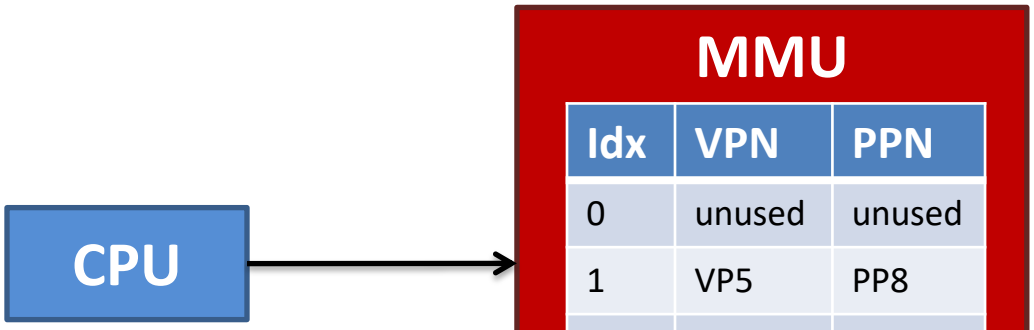
Access pattern:
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

TLB:
access **0x1234**, TLB Miss, cache VP1<->PP3

Both 0x1234 and 0x5234 go to entry 1

Speedup Address Translation



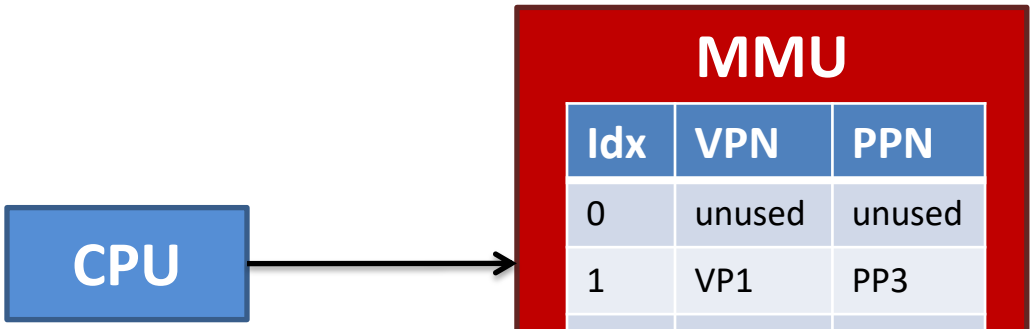
Access pattern:
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

TLB:
access **0x1234**, TLB Miss, cache VP1<->PP3
access **0x5234**, TLB Miss, evict VP1<->PP3, cache VP5<->PP8

Both 0x1234 and 0x5234 go to entry 1

Speedup Address Translation



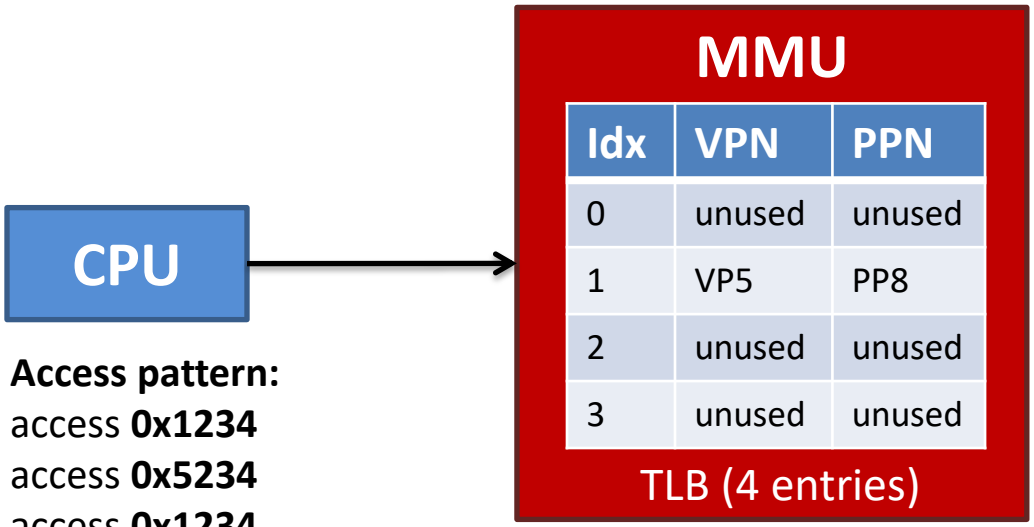
Access pattern:
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

TLB:
access **0x1234**, TLB Miss, cache VP1<->PP3
access **0x5234**, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
access **0x1234**, TLB Miss, evict VP5<->PP8, cache VP1<->PP3

Both 0x1234 and 0x5234 go to entry 1

Speedup Address Translation



Access pattern:
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

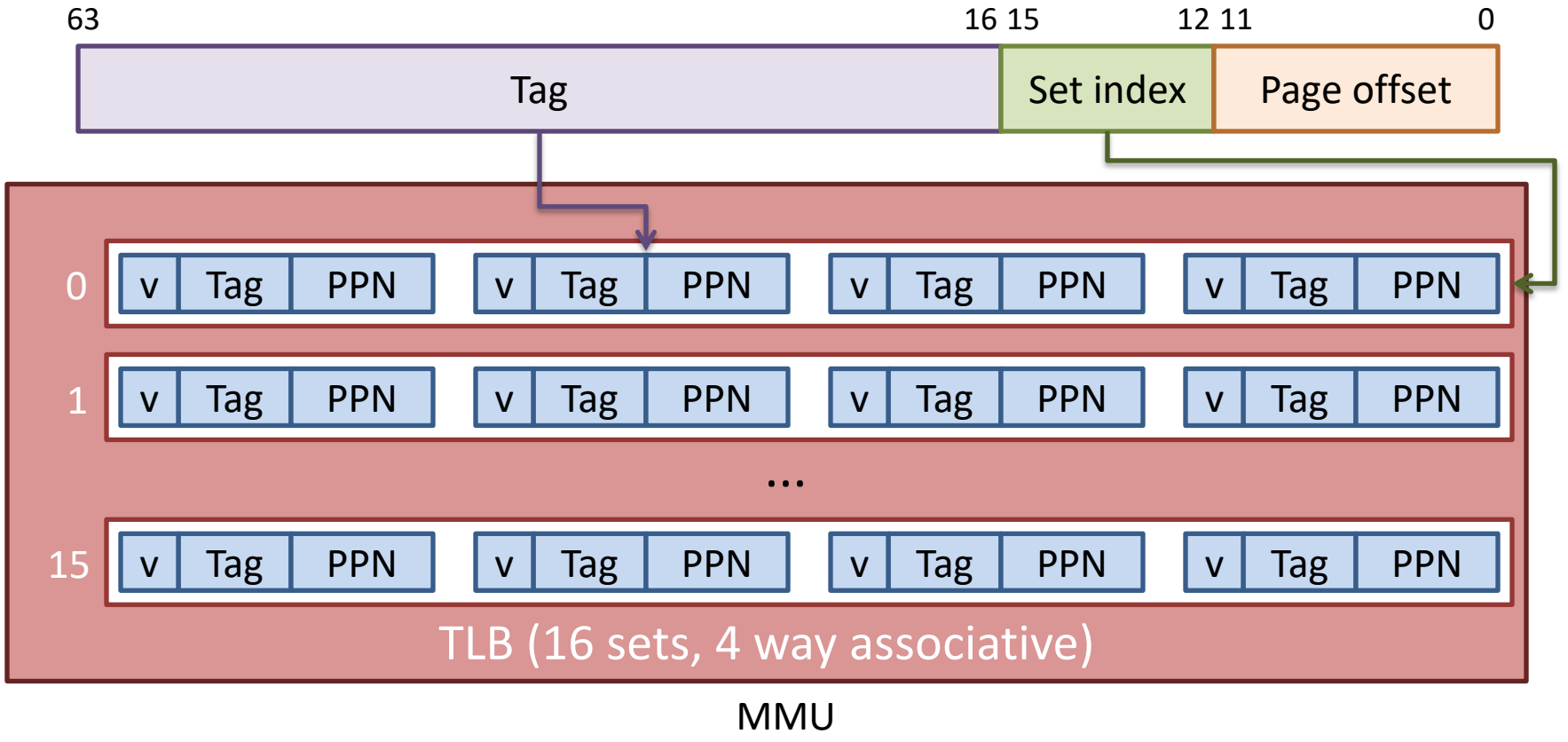
1. Calculate VPN
   a. VPN = VA >> 12
   b. Offset = VA & 0xfff
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. Yes (TLB hit)
      PA = TLB[Index].PPN + Offset
   d. No (TLB miss)
      Go though page table to get PPN
      Buffer the result in TLB

TLB:
access **0x1234**, TLB Miss, cache VP1<->PP3
access **0x5234**, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
access **0x1234**, TLB Miss, evict VP5<->PP8, cache VP1<->PP3
access **0x5234**, TLB Miss, evict VP5<->PP8, cache VP1<->PP3

TLB eviction due to conflict!

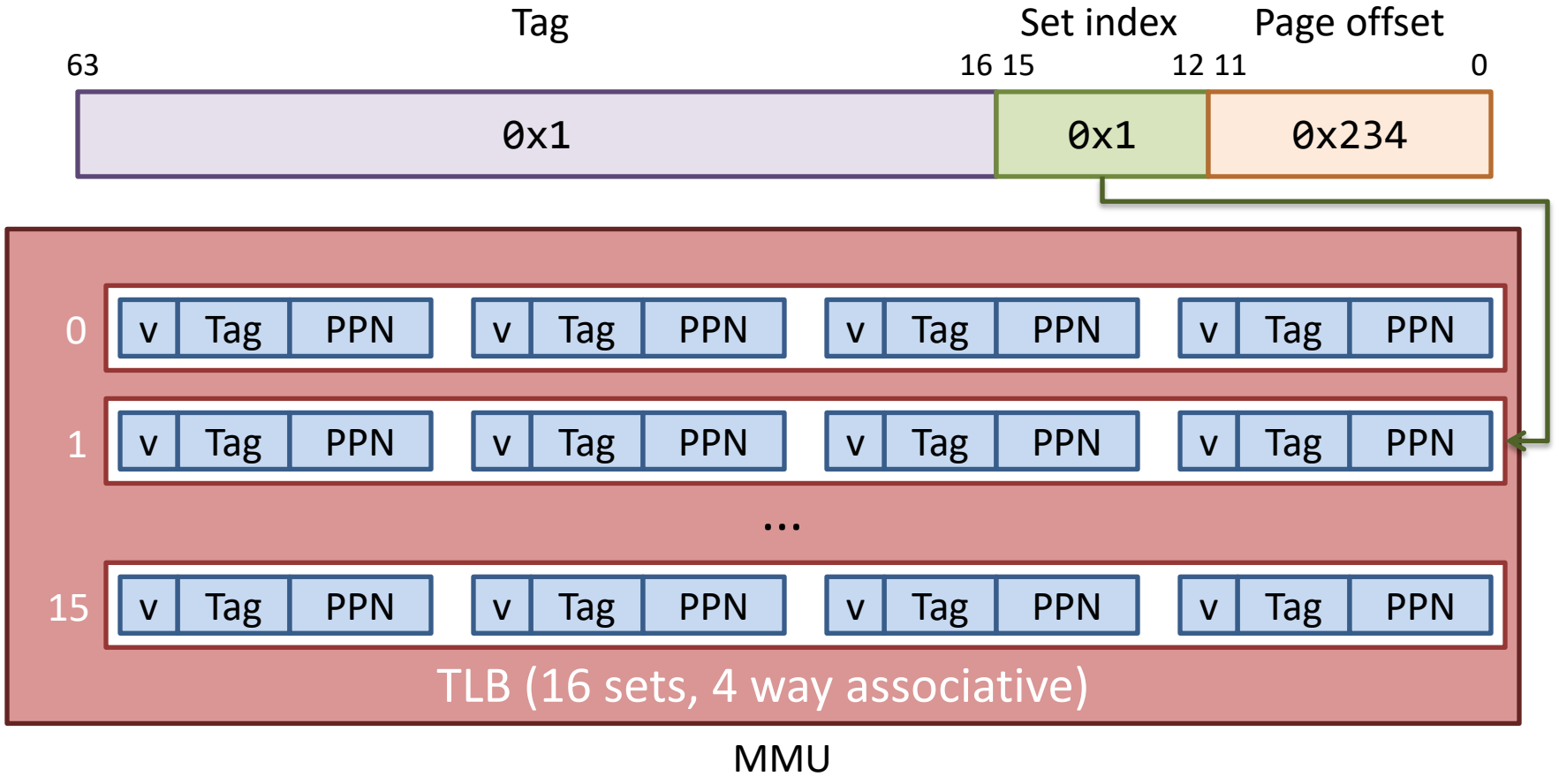
Both 0x1234 and 0x5234 go to entry 1

Multi-set associative TLB



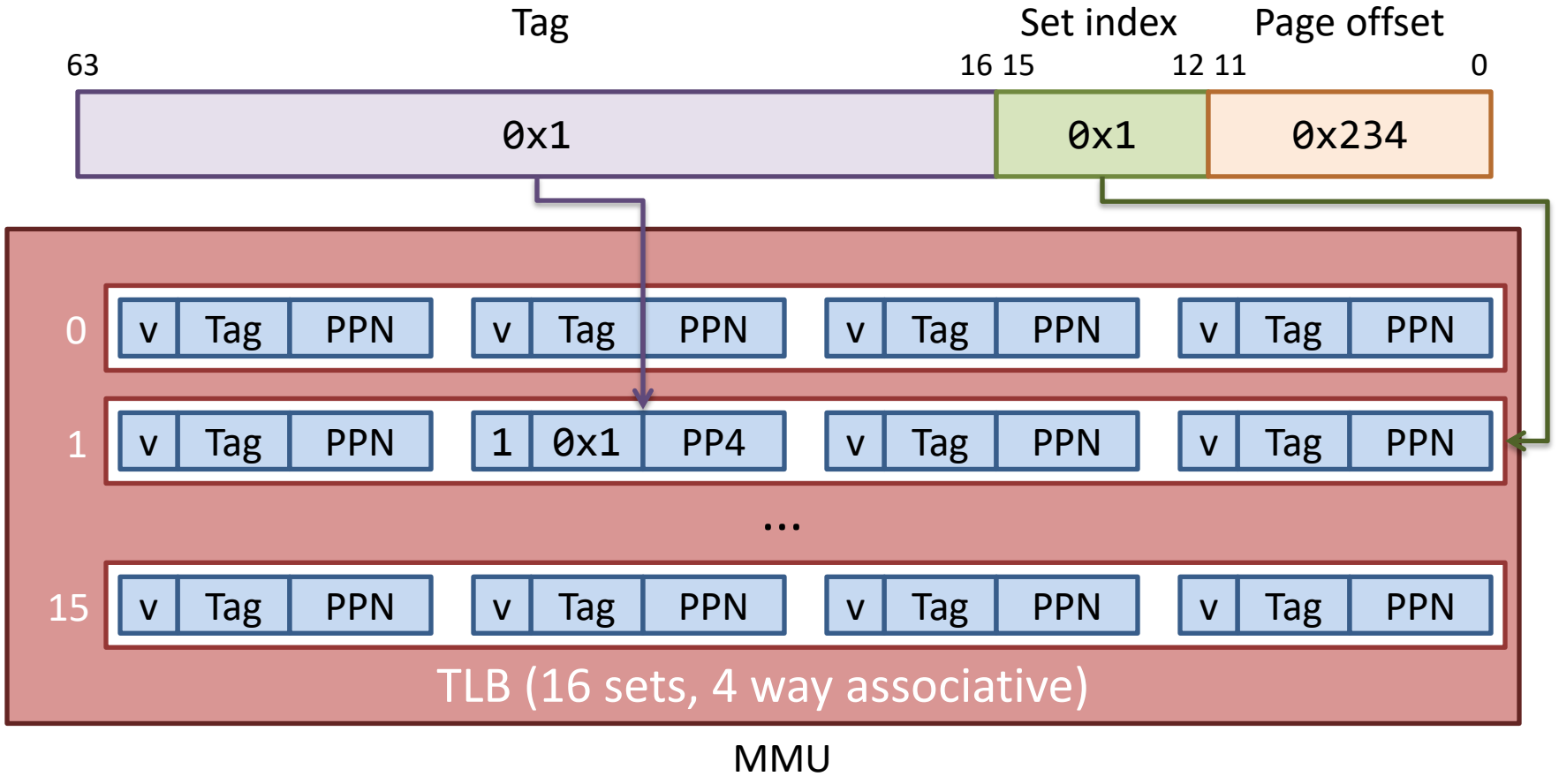
Example

→ access **0x11234, TLB Miss**
access **0x21234**
access **0x11234**
access **0x21234**



Example

→ access **0x11234, TLB Miss, cache the translation result**
access **0x21234**
access **0x11234**
access **0x21234**



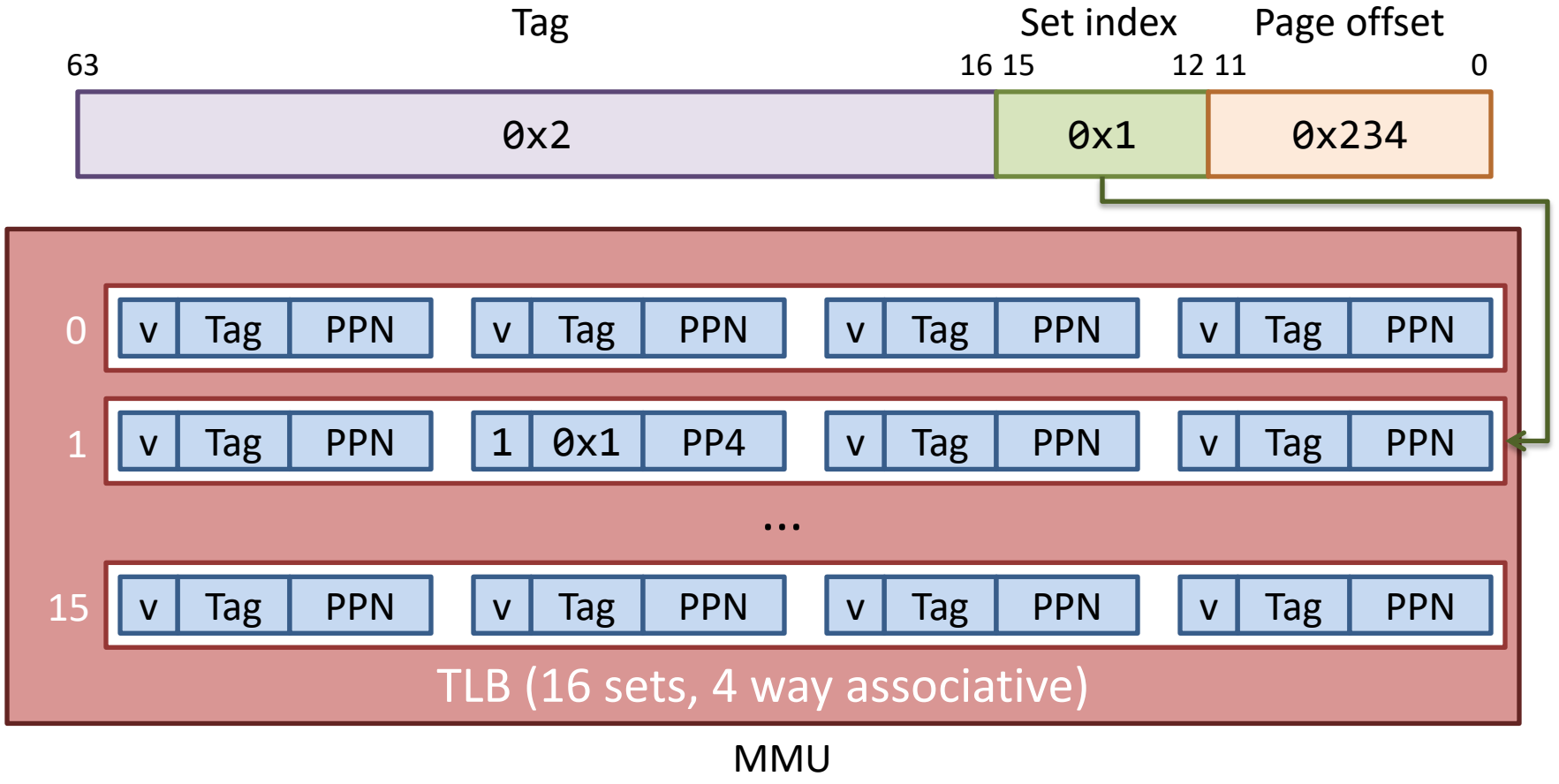
Example

access **0x11234, TLB Miss, cache the translation result**

→ access **0x21234, TLB Miss**

access **0x11234**

access **0x21234**



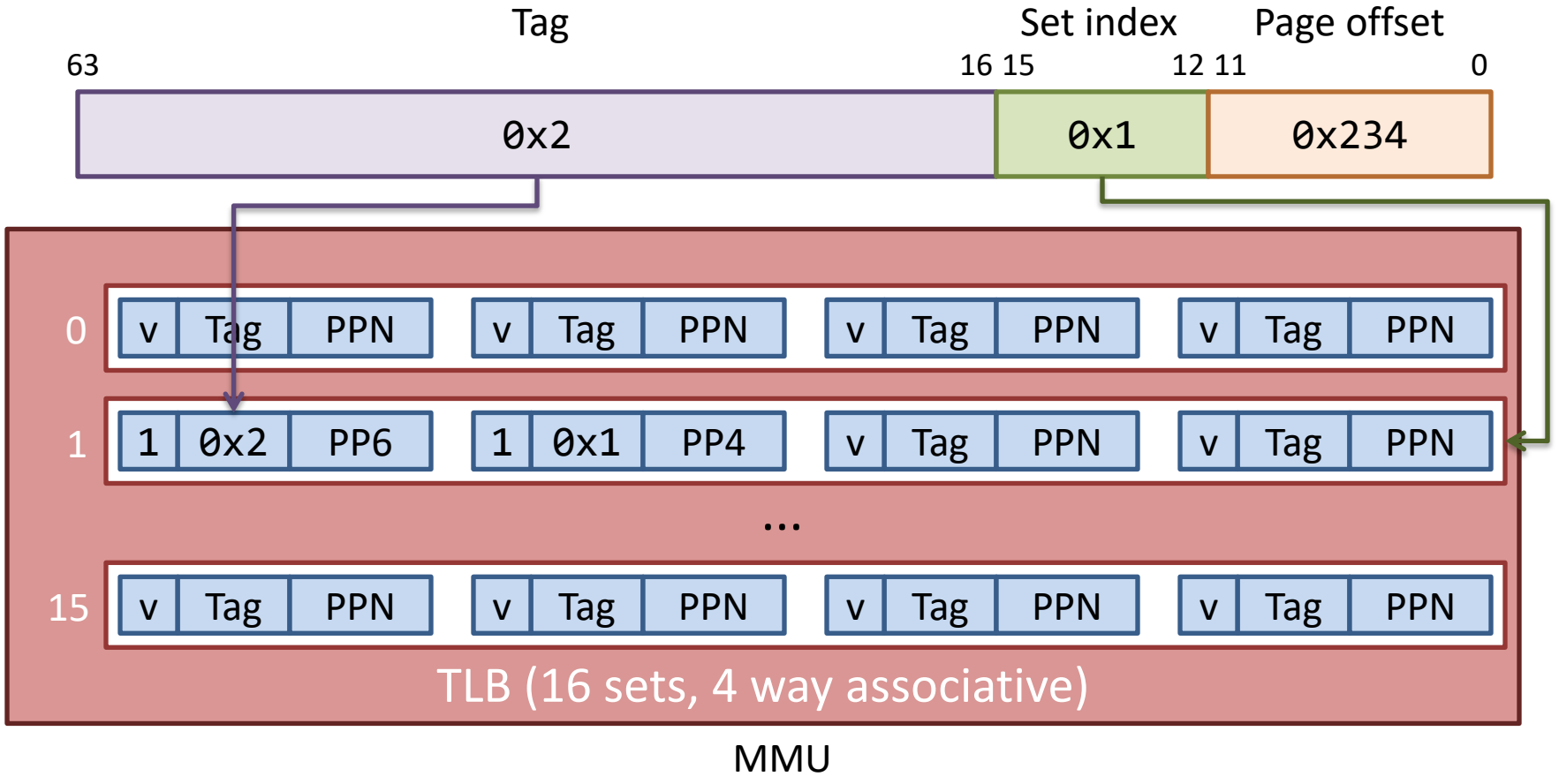
Example

access **0x11234, TLB Miss, cache the translation result**

→ access **0x21234, TLB Miss, cache the translation result**

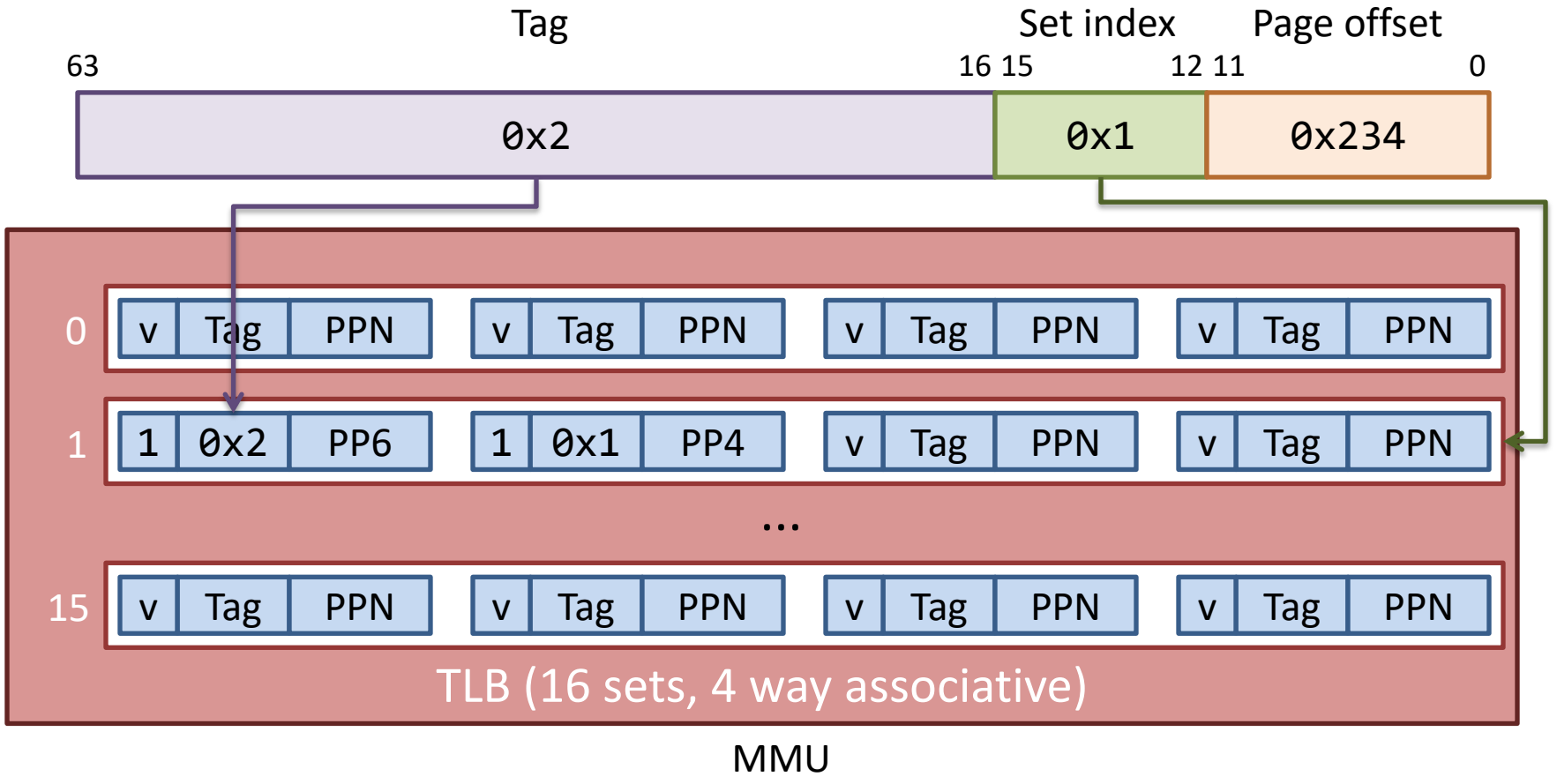
access **0x11234**

access **0x21234**



Example

access **0x11234, TLB Miss, cache the translation result**
access **0x21234, TLB Miss, cache the translation result**
access **0x11234, TLB Hit**
access **0x21234, TLB Hit**



Memory and Cache

Question

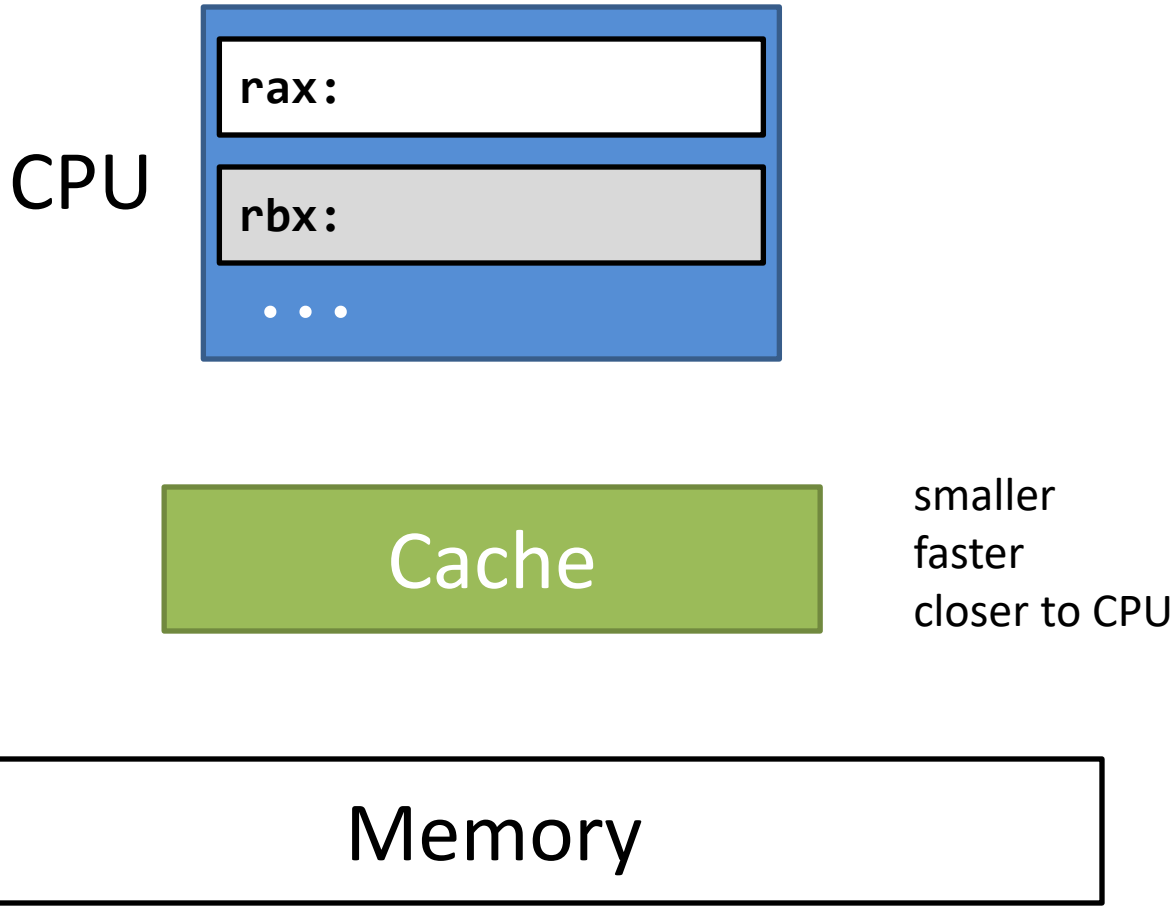
- How many memory accesses are needed to execute
  - `movq (%rax), %rbx` 1 memory access (~100 CPU cycles)
  - `addq %rax, %rbx` 0 memory accesses (~1 CPU cycles)

- How to reduce the cost of memory accesses?

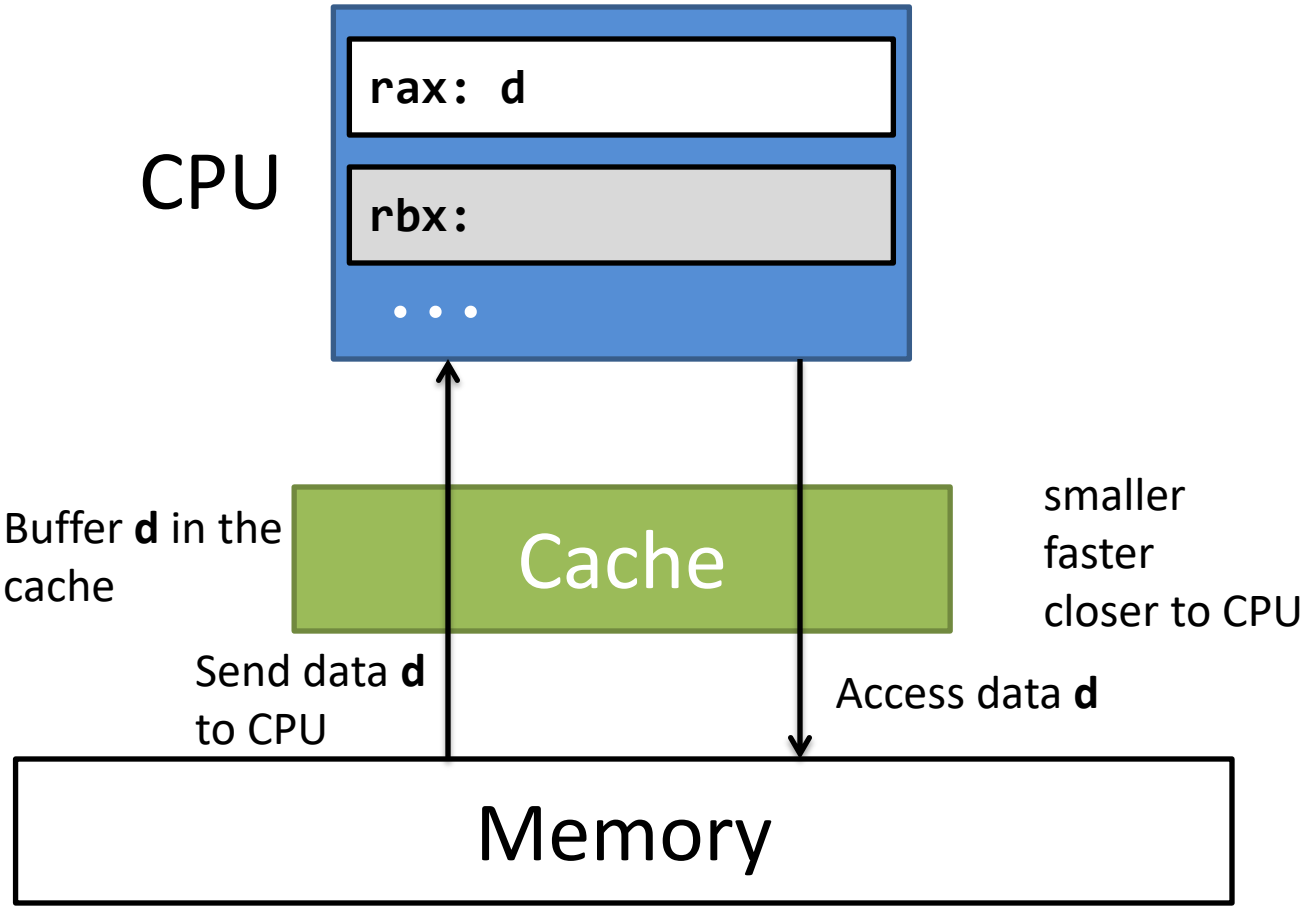
Principle of Locality

- **Temporal locality**
  - If memory location x is referenced, then x will likely be referenced again in the near future.
- **Spatial locality**
  - If memory location x is referenced, then locations near x will likely be referenced in the near future.
- **Idea**
  - Buffer recently accessed data in cache close to CPU

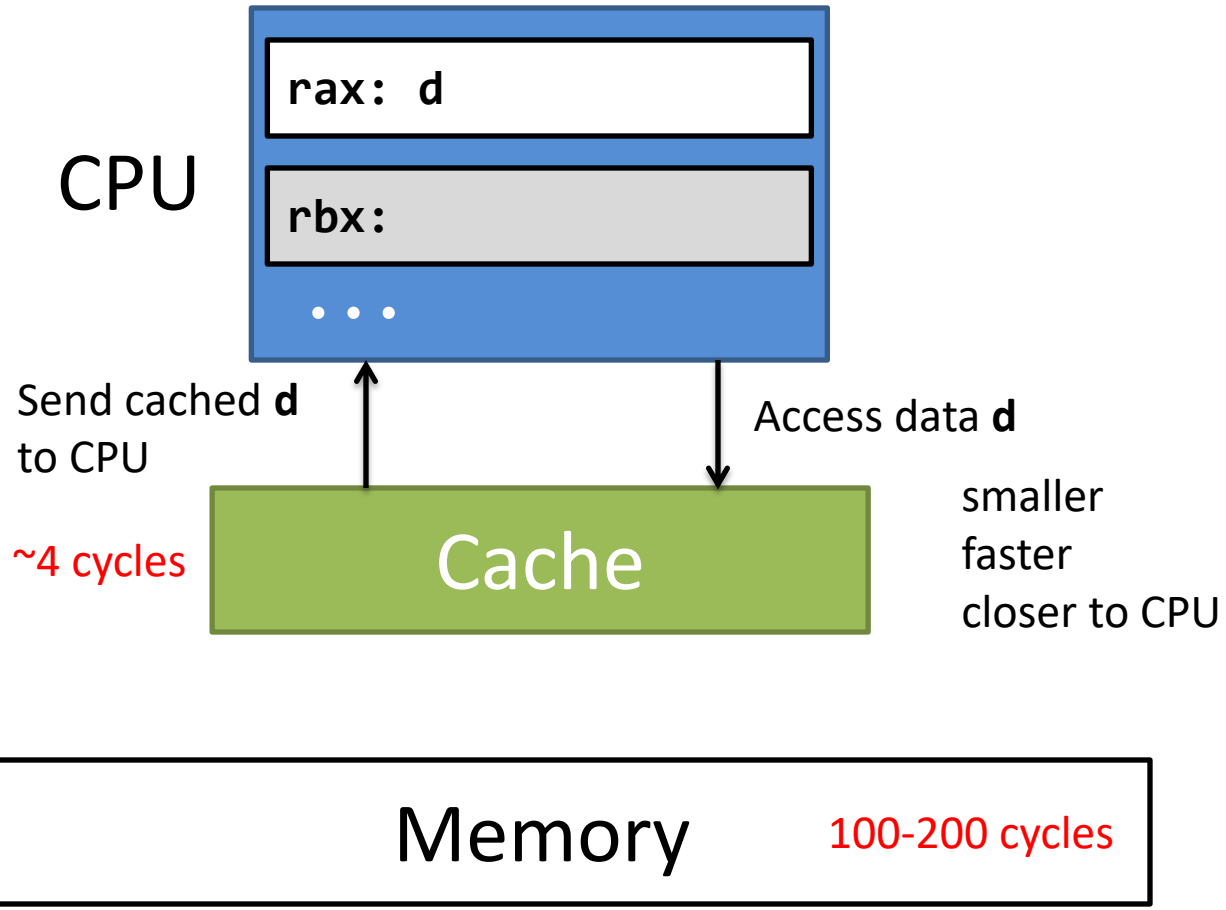
Basic Idea - Caching



Basic Idea - Caching

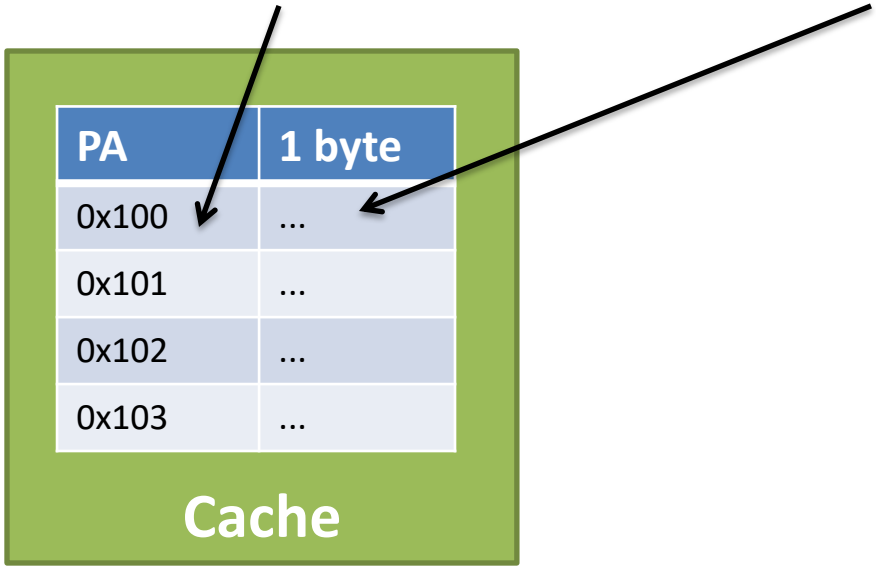


Basic Idea - Caching



Intuitive implementation

- Caching at byte granularity:
  - Search the cache for each byte accessed
    - `movq (%rax), %rbx` → checking 8 times
- High bookkeeping overhead
  - each cache entry has 8 bytes of address and 1 byte of data



Caching at Block Granularity

Solution:

- Cache one block (cache line) at a time.
- A typical cache line size is 64 bytes

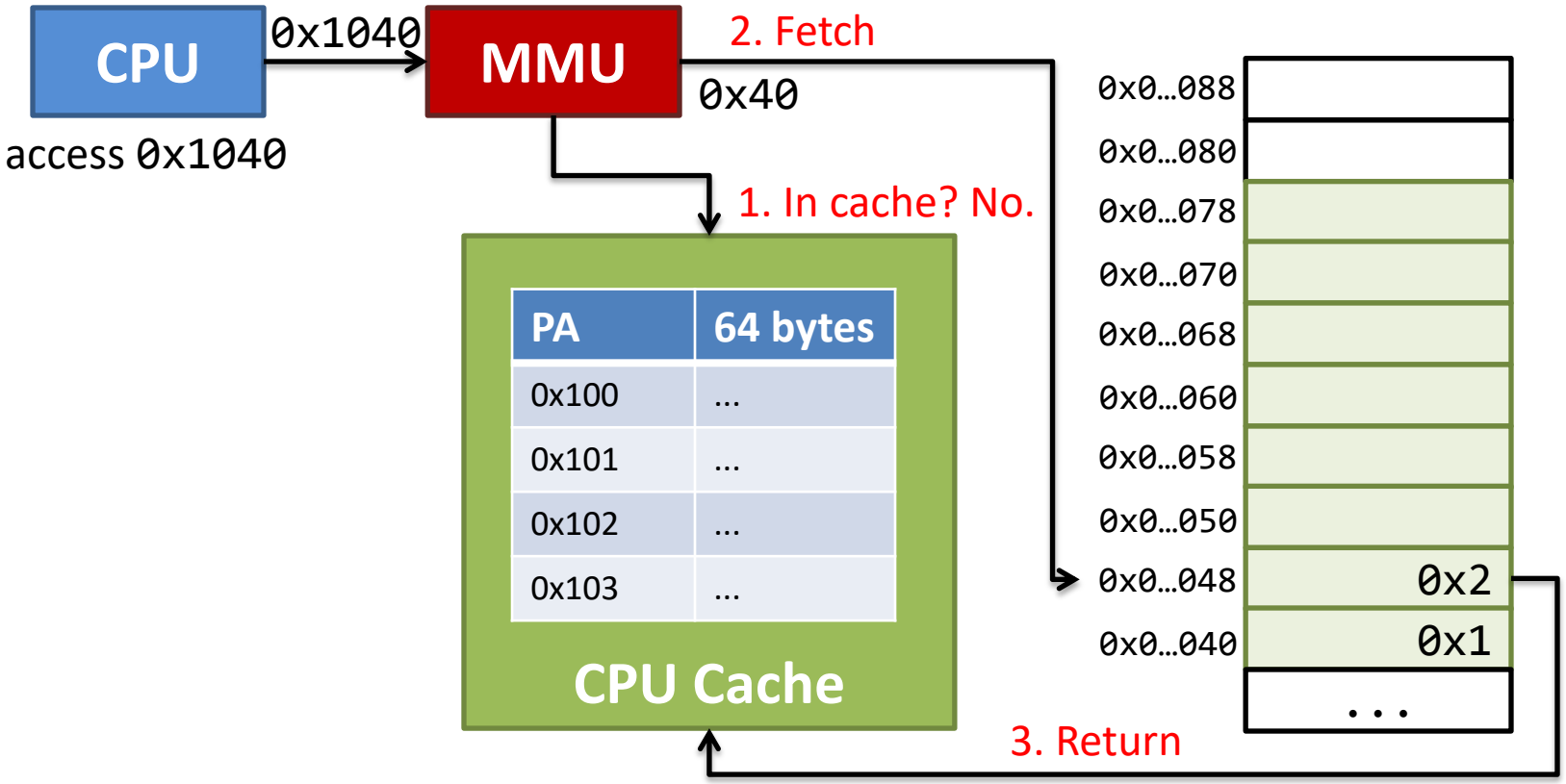
Advantage:

- Lower bookkeeping overhead
  - A cache line has 8 byte of address and 64 byte of data
- Exploits spatial locality
  - Accessing location x causes 64 bytes around x to be cached

Direct-mapped cache

Caching at block granularity

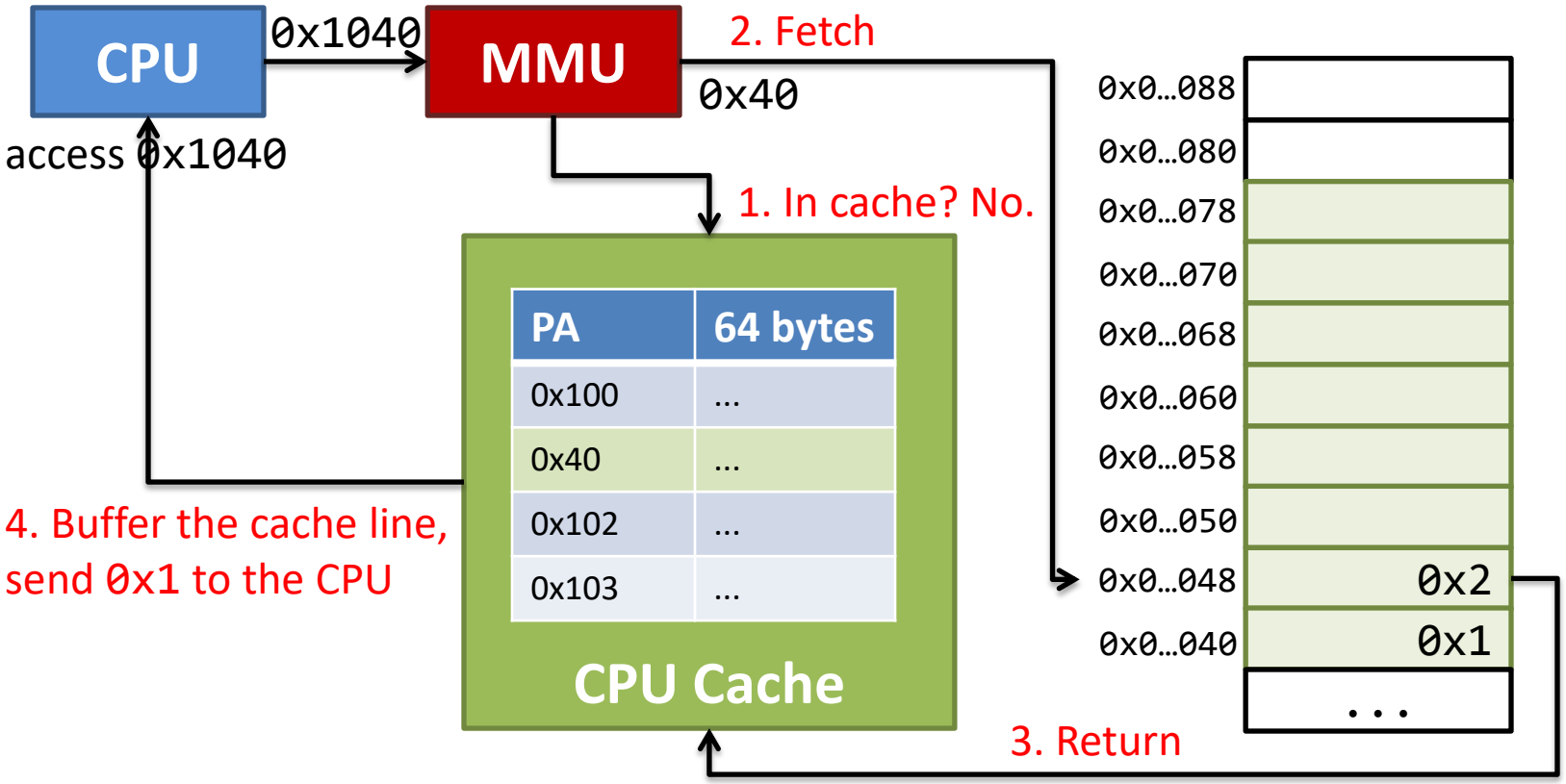
- Each cache line has 64 bytes



Direct-mapped cache

Caching at block granularity

- Each cache line has 64 bytes



Direct-mapped cache

Caching at block granularity

- Each cache line has 64 bytes