

Exercise Session 12

Systems Programming and
Computer Architecture

Caches & Virtual Memory

Autumn Semester 2024

Disclaimer

- **Website:** n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub:** github.com/falkbe
- **Kahoots:** now on website n.ethz.ch/~falkbe/
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page
- Information from the exercise session is partly taken from ***Digital Design and Computer Architecture*** by David Money Harris, Sarah L. Harris

Agenda

- Exceptions and Kernel
- Caches
- Virtual memory
- Quiz, Exam on VM and Caches
- Caches
- Virtual Memory
- Address Translation
- Preview Assignment 10
- Quiz Caches

Exceptions and the Kernel

Remark

- Exceptions as taught in lecture are **exam relevant**
- **The** following slides (in particular the kernel slides) are just thought to give you **more context** on what “the kernel” is
- **You do not need to know kernel stuff in this depth** (again, only as much as taught in the lecture)
- The following slides are from the **Computer Systems Course**

Recall from Lecture

The Kernel

- Most operating systems have a **kernel**
= the part of the OS that runs in kernel mode
- Think of the kernel as:
 - A set of trap handling functions (always)
 - A set of threads in a special address space (sometimes)
 - Code to create the illusion of user-space processes (always)
- Many ways to structure the kernel and OS
 - Microkernels, monolithic kernels, multikernels, etc.
 - See the OS course next year...
- **What does this actually mean?**

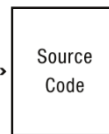
Exceptions and the Kernel

A Problem

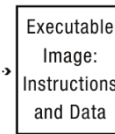
- Assume one application
 - For now...



Edits

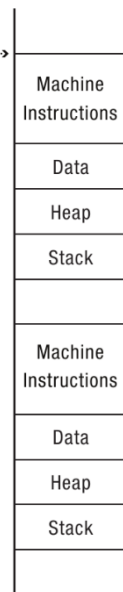


Compiler



Operating System Copy

Physical Memory



- How can we prevent the application:
 - Corrupting the OS?
 - Accessing hardware that it shouldn't?

Exceptions and the Kernel

Thought Experiment

- How can we implement execution with limited privilege?
 - Execute each program instruction in a simulator
 - If the instruction is permitted, do the instruction
 - Otherwise, stop the process
 - Basic model in Javascript and other interpreted languages
- How do we go faster?
 - Run the unprivileged code directly on the CPU!

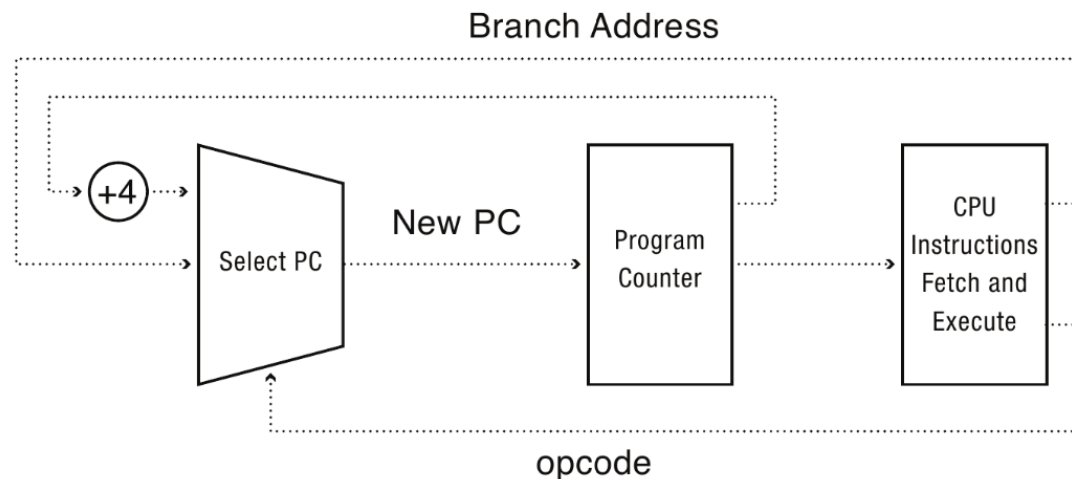
Exceptions and the Kernel

Hardware Support: Dual-Mode Operation

- Kernel mode
 - Execution with the full privileges of the hardware
 - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
 - Code here must be carefully written!
- User mode
 - Limited privileges
 - Only those granted by the operating system kernel

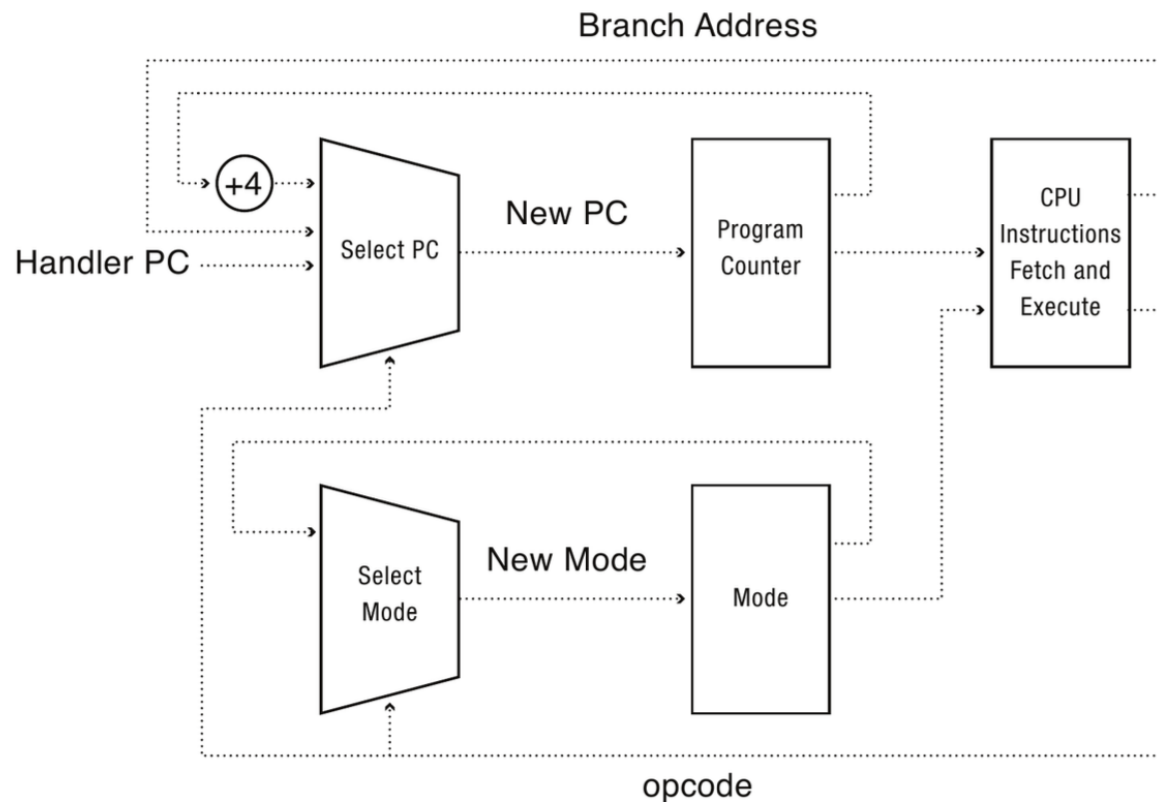
Exceptions and the Kernel

A simple model of a CPU



Exceptions and the Kernel

CPU with dual-mode operation (or more...)

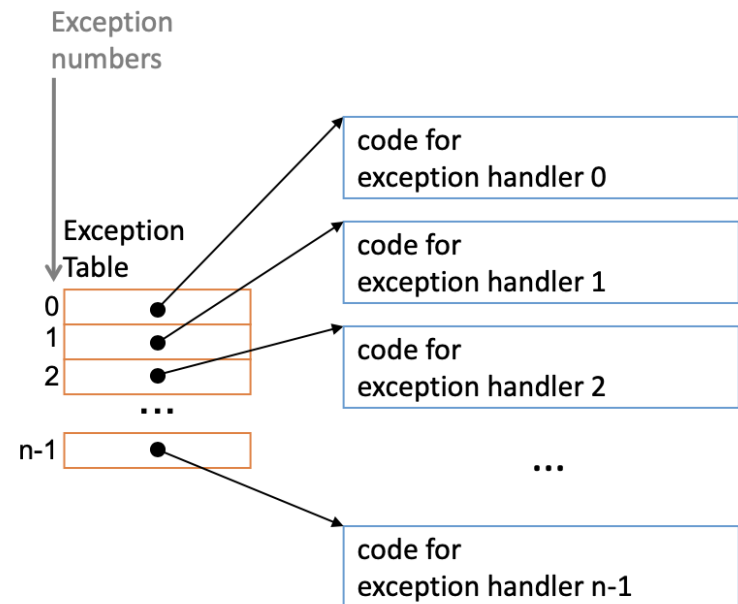


Computer Systems 2024, Ch. 3: The Kernel

Exceptions and the Kernel

Basic primitive: processor exceptions

- When an exception occurs:
 - Finish executing current instruction
 - Switch mode from user to kernel
 - Look up exception cause in the exception vector table
 - Jump to this address
- And possibly:
 - Save registers (or switch banks)
 - Switch page table (usually not)



Exceptions and the Kernel

Types of Exceptions

- A *synchronous* exception occurs as a result of executing an instruction.
- An *asynchronous* exception occurs as a result of events that are external to the processor.

Type of exception	Cause	Async/Sync
Interrupt	Signal from I/O device	Async
Trap	Intentional exception	Sync
Fault	Potentially recoverable error	Sync
Abort	Nonrecoverable error	Sync

Async Exceptions in a Nutshell



(meme stolen from DINFK Discord)

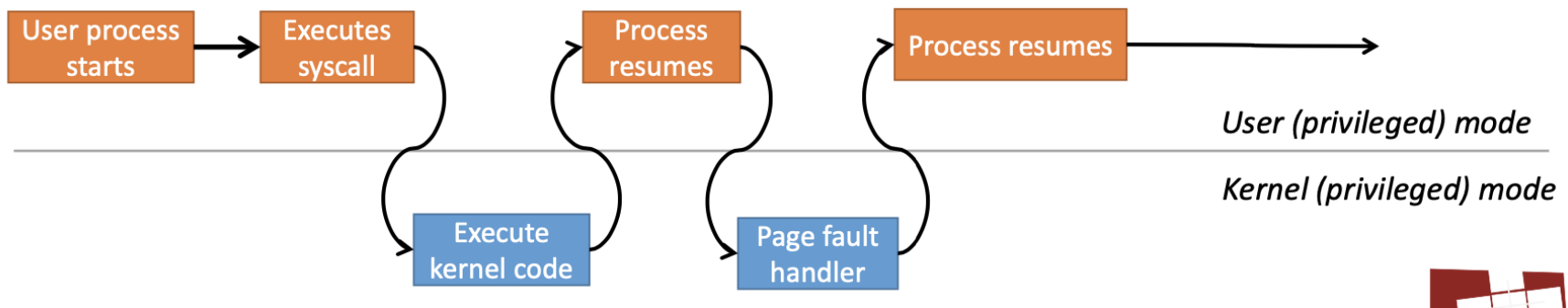
executing an instruction.
of events that are

	Async/Sync
	Async
	Sync
	Sync
	Sync

Exceptions and the Kernel

Conventional perspective:

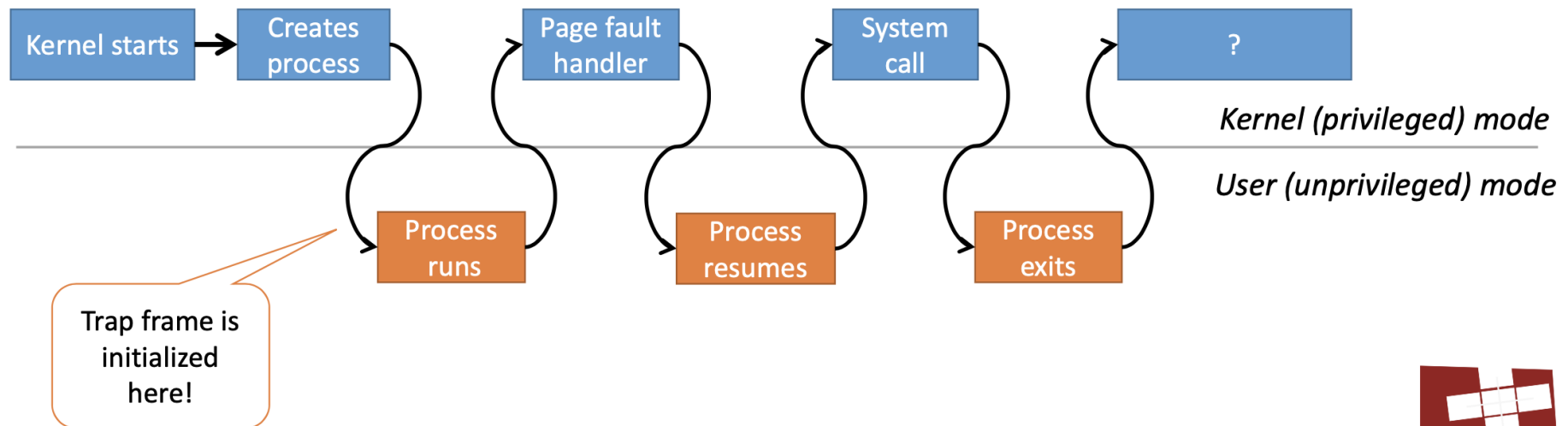
- User programs run until the kernel needs to
 - System call
 - Page fault
 - Interrupt, etc.



Exceptions and the Kernel

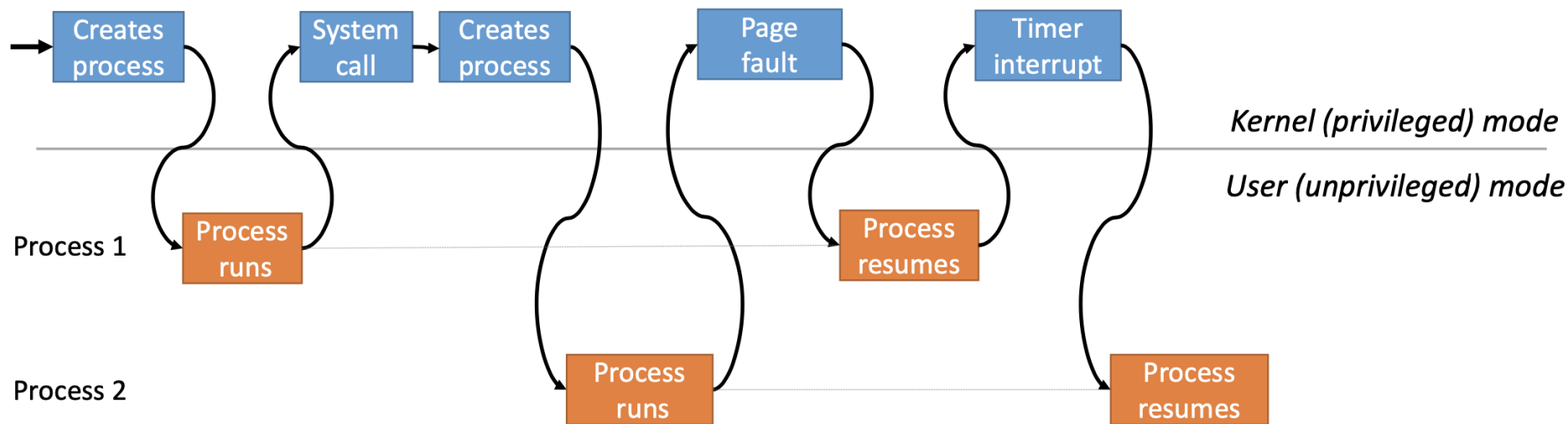
Alternative perspective:

- Kernel runs, calls sandboxed user applications, then retakes control.



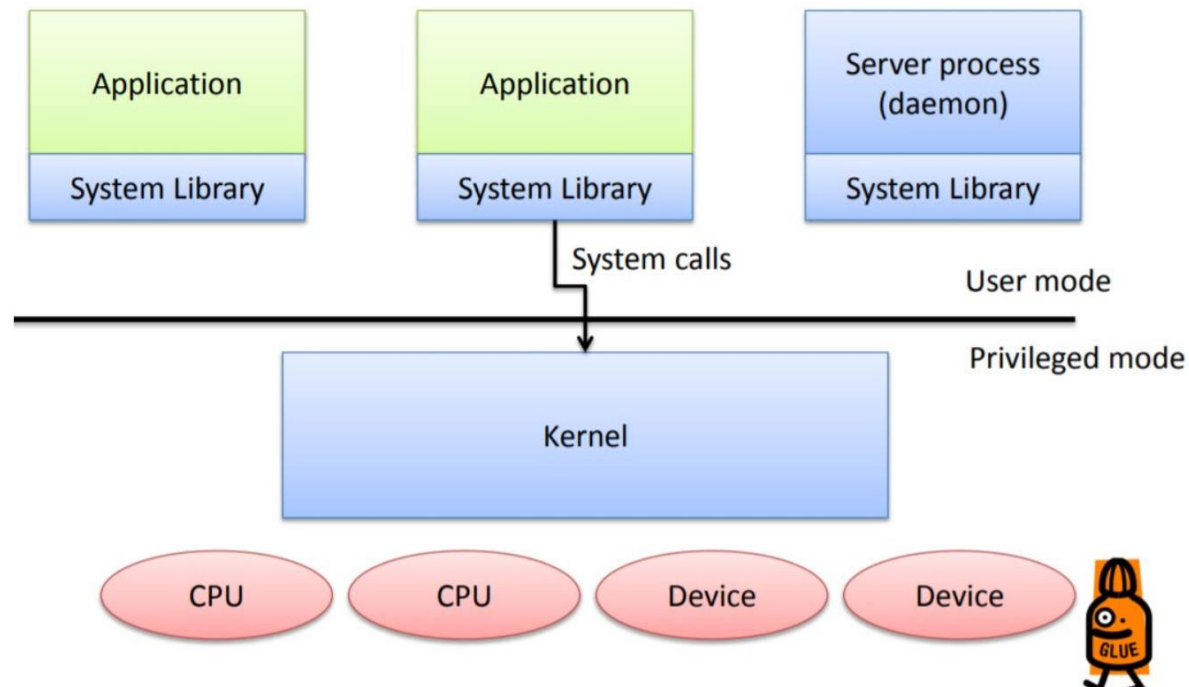
Exceptions and the Kernel

The illusion of multiple computers



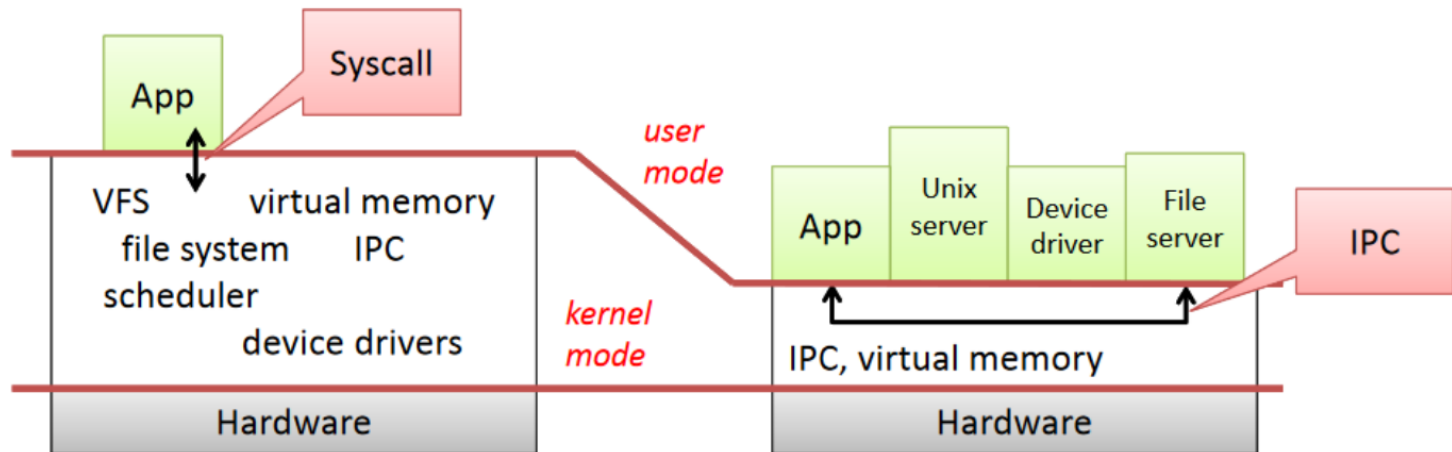
Exceptions and the Kernel

General model of OS structure



Exceptions and the Kernel

Monolithic kernels vs. Microkernels



- Monolithic OS
 - lots of privileged code
 - services invoked by syscall
- Microkernel OS:
 - little privileged code
 - services invoked by IPC
 - “horizontal” structure

Remark

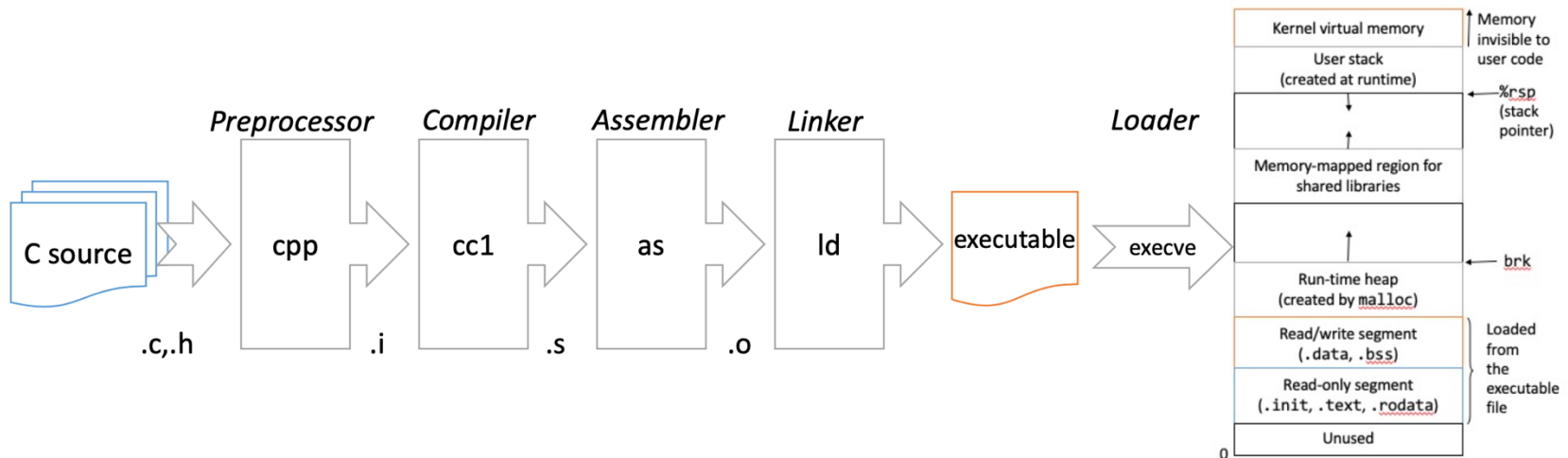
- **Interested?** Take **Computer Systems** core subject in 5th semester

Caches

Where are we in the course

- **Compilation pipeline:** from C source code to assembly, to the executable (and how this is layed out "in memory")

Recall: how C code runs as a process on CPU



Where are we in the course

- Computer Architecture: processor design

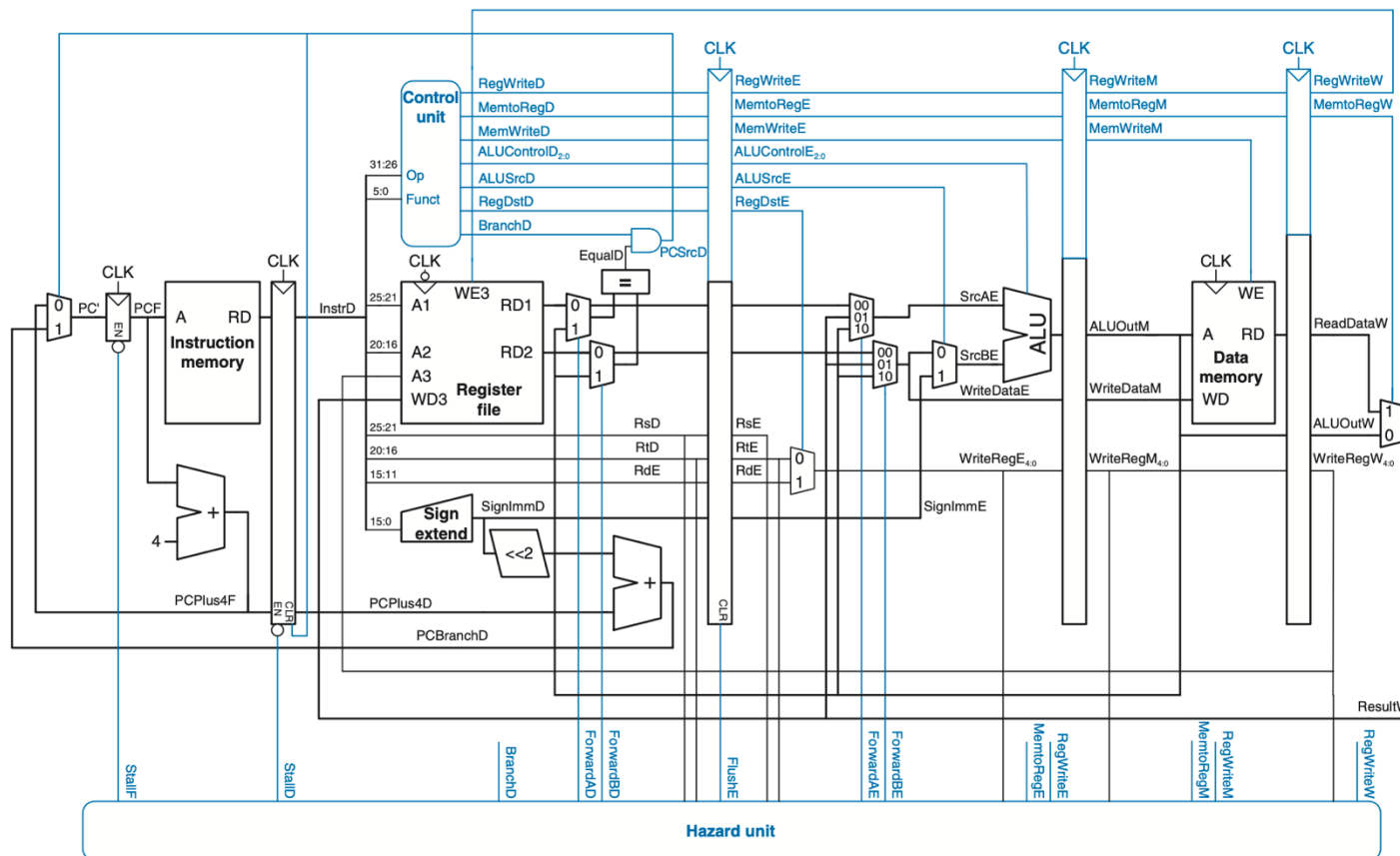
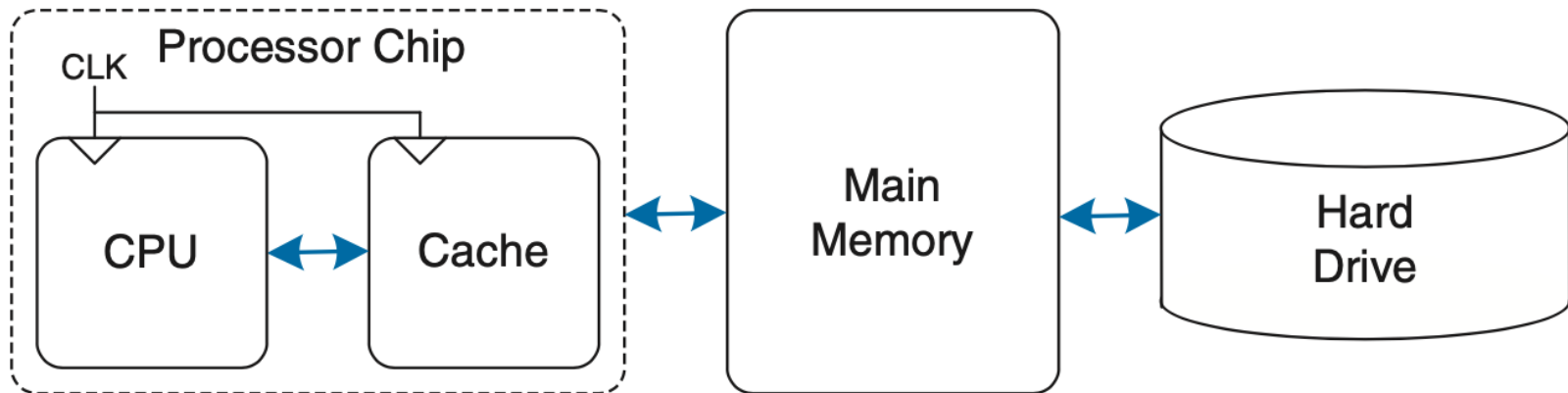


Figure 7.58 Pipelined processor with full hazard handling

Where are we in the course

- Currently: **memory hierarchy** (caches, vm)



Where are we in the course

- Later: **embedded devices**

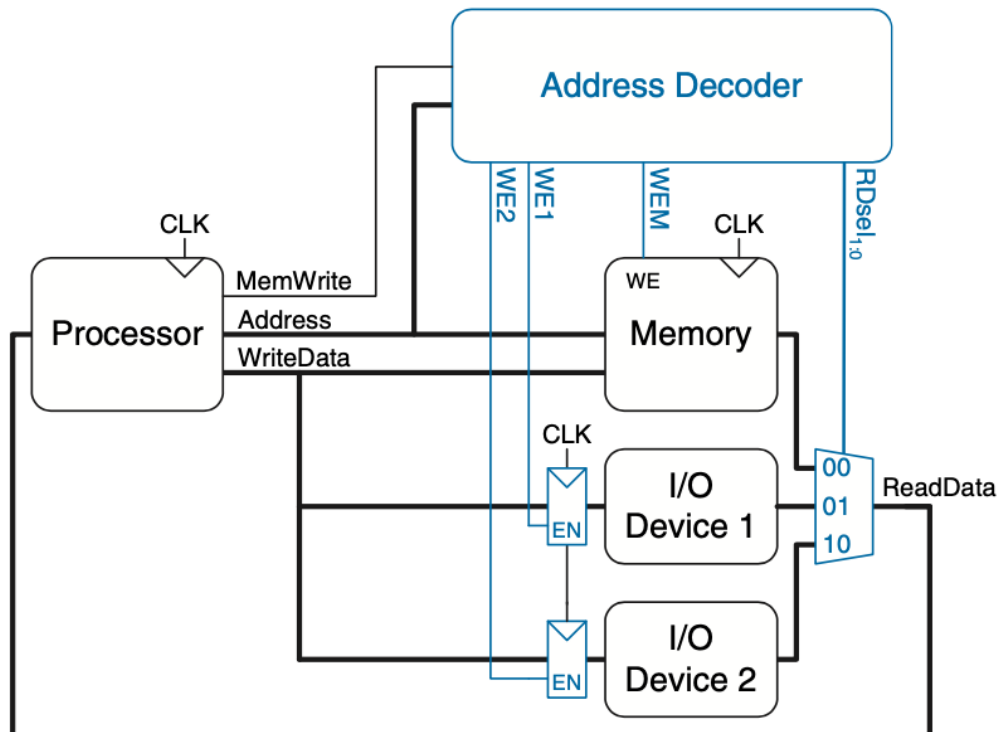


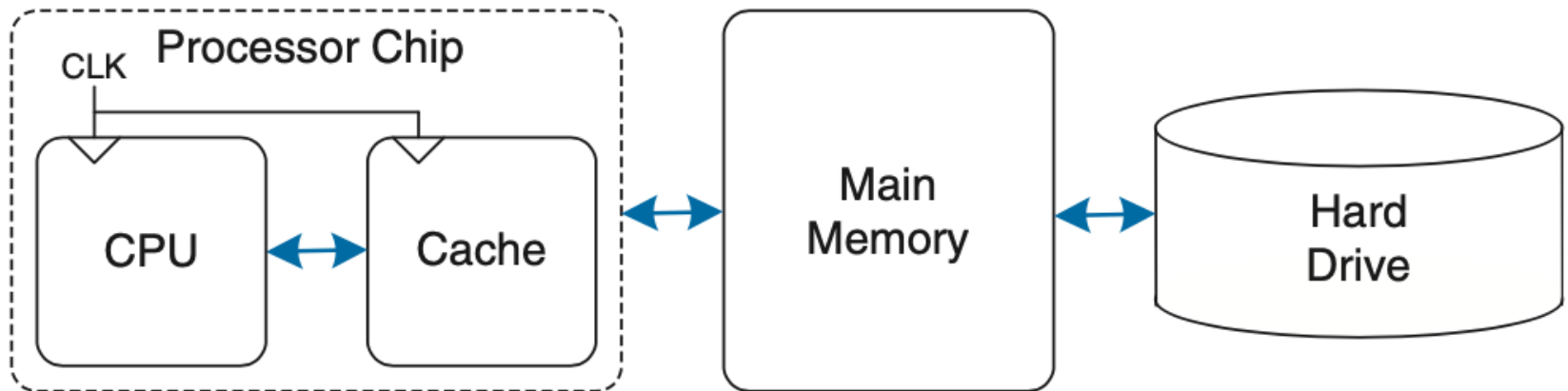
Figure 8.28 Support hardware for memory-mapped I/O

Caches

- **Analogy for Memory Hierachy:** Library
- **Cubical (Cache):** Keeping books we recently used or likely to use in the future at our cubicle (based on temporal and spatial locality)
 - **Temporal Locality:** if we used the book recently, we are likely to use it again
 - **Spatial Locality:** Interested in one book, so likely to be interested in other books of the same shelf
- **Shelves (Main Memory):** Keeps most used books in shelf
- **Basement (Disk):** keeps lesser-used book in deep storage in the basement

Caches

- **Memory hierarchy** graphically
- Processor seeks data
 - 1. looks in cache, if not here then
 - 2. looks in main memory, if not here
 - 3. fetches data from disk/hard drive



Caches: Data held in the cache

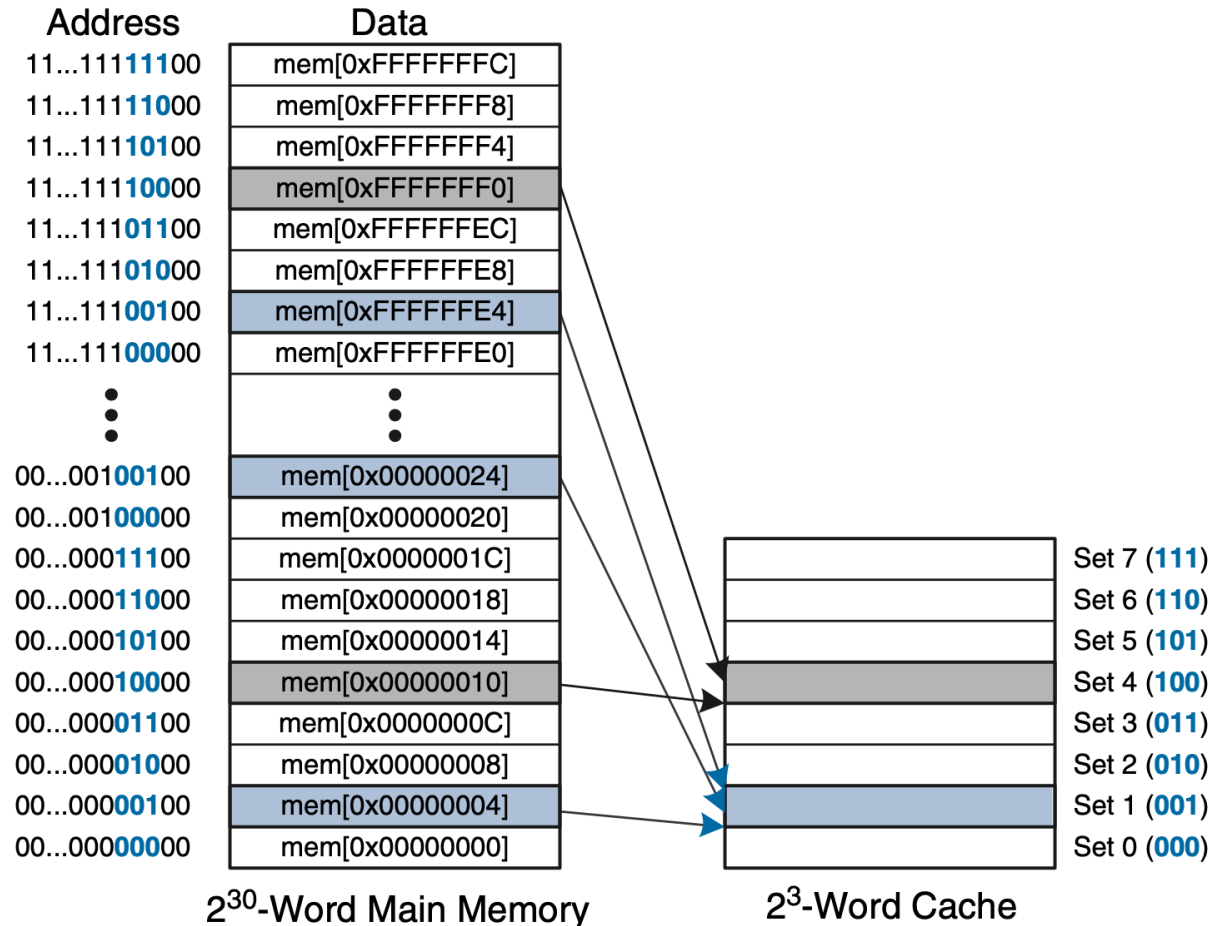
- **Caches exploit temporal and spatial locality**
- **Temporal locality:** processor is likely to access data again soon, if it has accessed it recently [local variables]
 - => If data is not in cache, processor fetches it from main memory and puts it into cache (subsequent requests hit in cache)
- **Spatial Locality:** when processor accesses piece of data, its likely to access nearby memory locations [array]
 - => Not just fetching one word, but several adjacent words, a “cache block”/”cache line”

Caches: How is data found

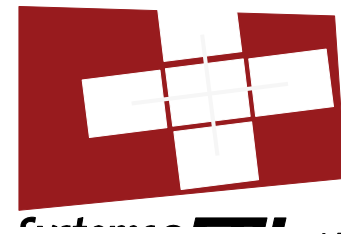
- **Cache:** Capacity **C**, Cache block **b**, Blocks **$B=C/b$**
 - *S sets (rows):* each set can hold block(s) of data
 - **Direct mapped** ($S=B$ sets): each block is in its own set
 - **N-way set associative** ($S=B/N$ sets): each set contains **N** blocks
 - **Fully associative** ($S=1$): one set containing all blocks
- **Mapping:** Relationship between address of data in memory and cache
 - Each memory address maps to exactly one set in the cache

Caches: Direct Mapped

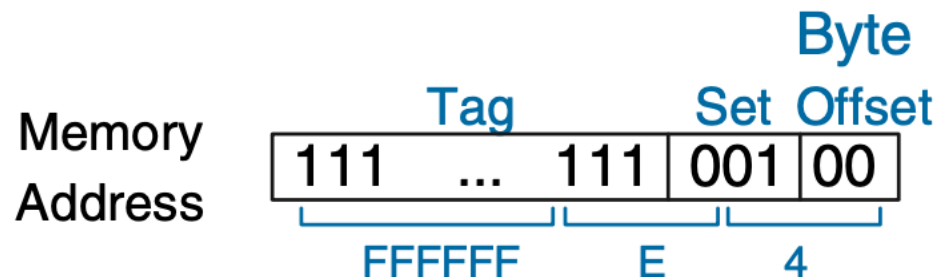
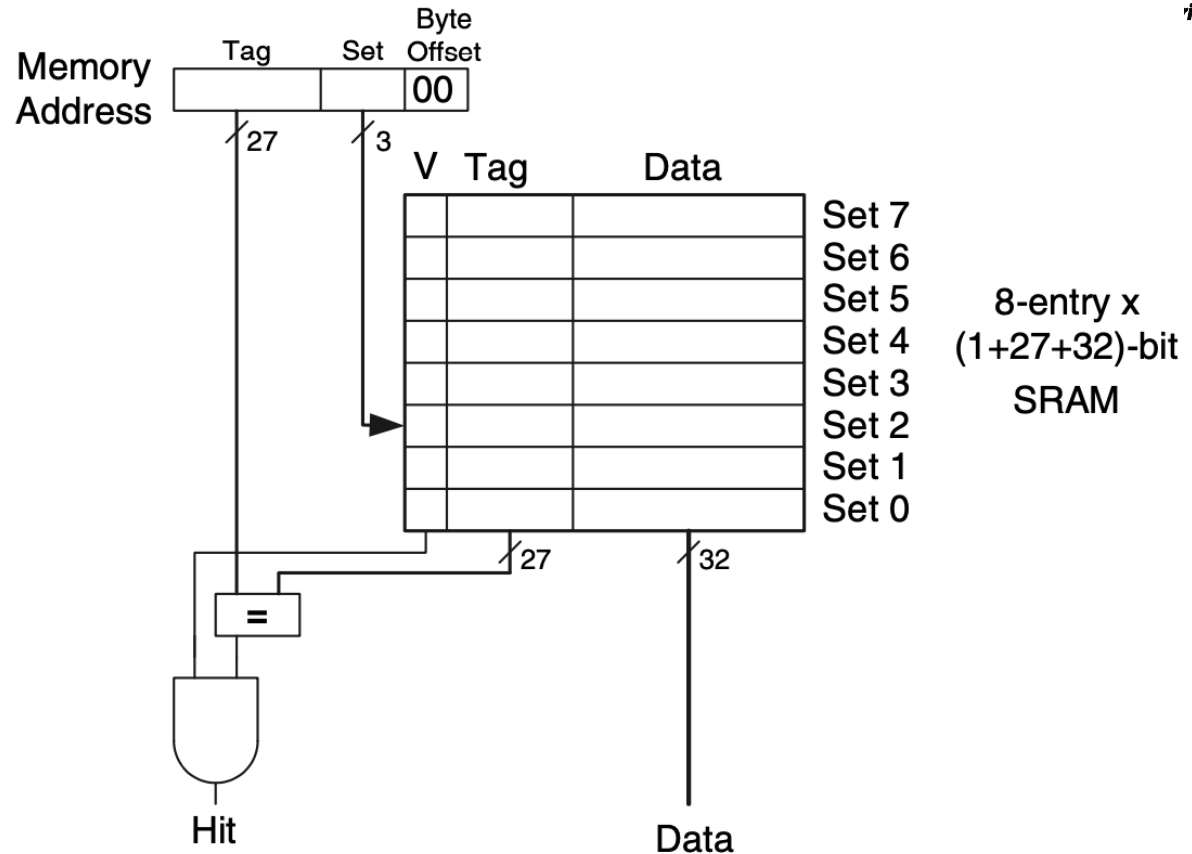
- **Direct mapped:** each set contains one block
- Bottom 2 bits 0 because its word (here 4 byte) aligned
- Next $\log_2(S)=3$ bits indicate set (mod 8)



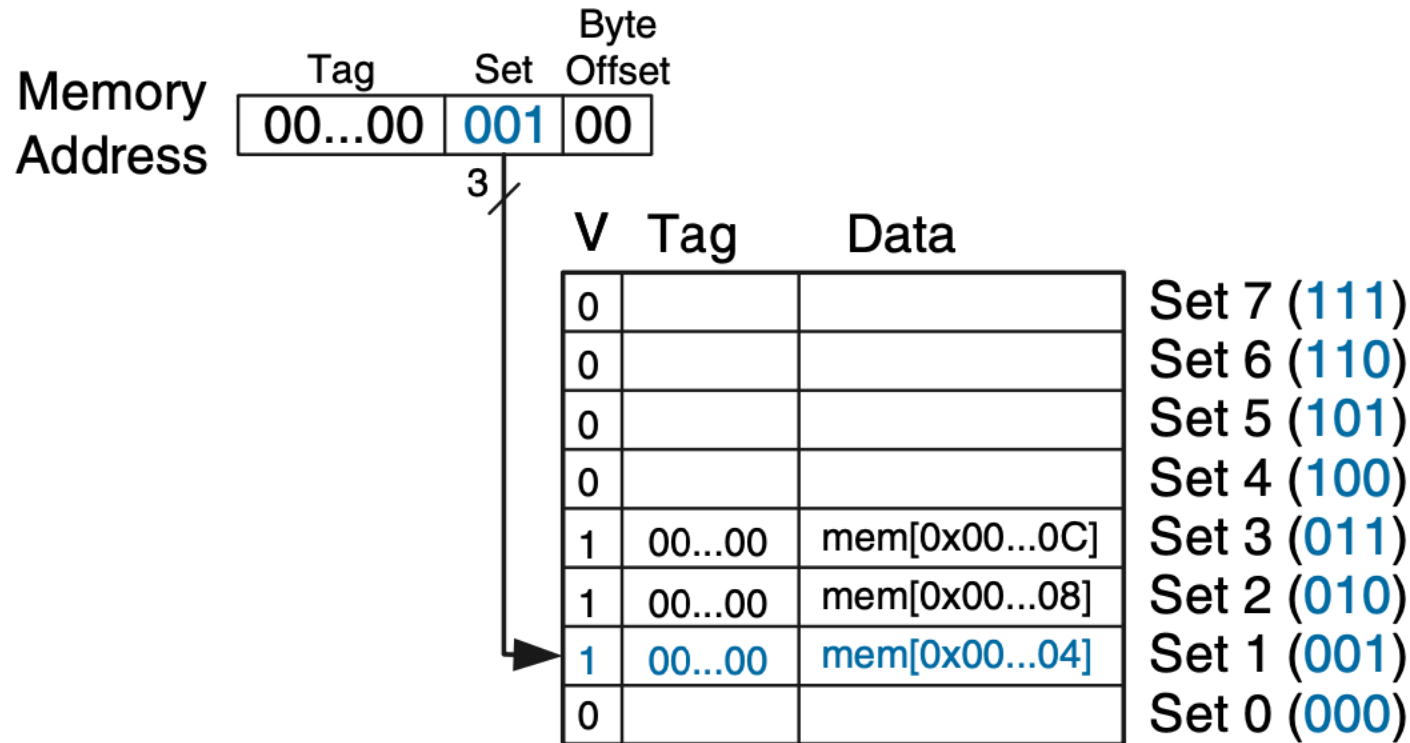
Caches: Direct Mapped



- **Byte offset:**
indicates byte within word
- **Set bits:**
indicate set in the cache
($\log_2(S)$ bits)
- **Tag bits:**
indicate memory address of data

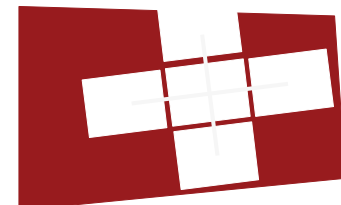


Caches: Direct Mapped

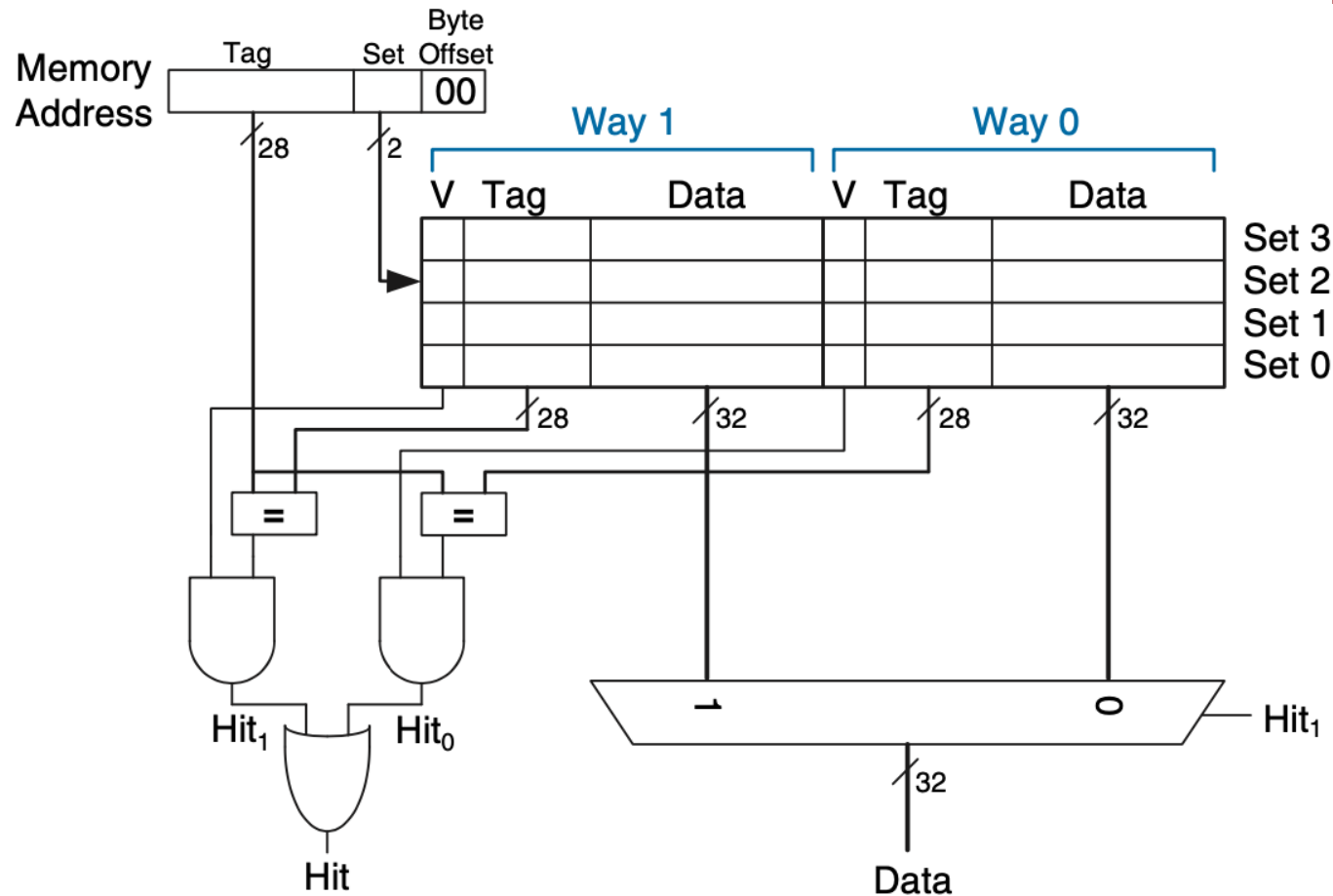


- If two memory address point to the same set: **conflict**
 - **One must be evicted** (removed from the cache)

Caches: N-way Set Associative



ms@ETH zürich



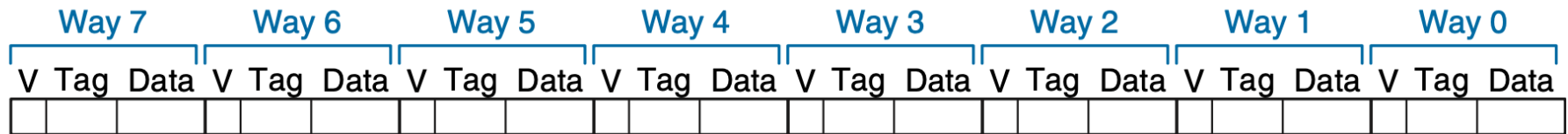
- **N-way**: every memory address still maps to a specific set, but can go into any of the n-ways inside this set (Here N=2)

Caches: N-way Set Associative

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...00	mem[0x00...24]	1	00...10	mem[0x00...04]	Set 1
0			0			Set 0

- **Advantage:** the higher the associativity, the less conflicts we have
 - Set associative caches generally have lower miss rate (only need to evict if both ways are full)

Caches: Fully Associative Cache



- **Fully associative:** B ways (number of blocks), i.e. no **conflict misses anymore**
- **Issue:** need a lot of comparators (compare 8 values in parallel)

Caches: Overview

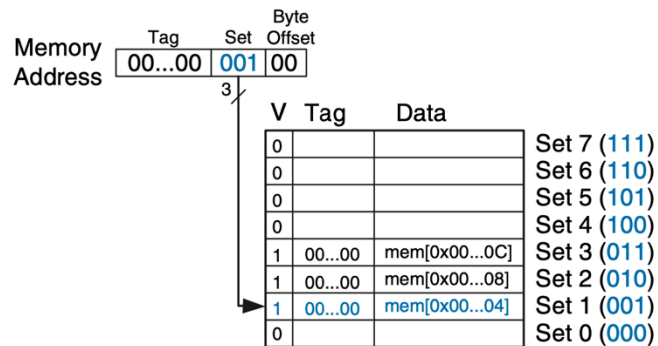
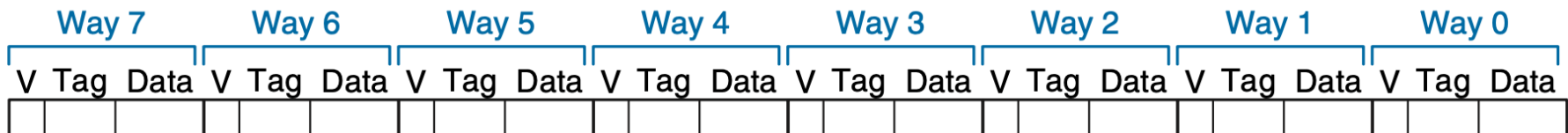
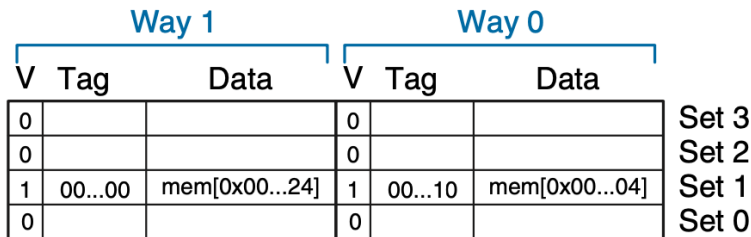


Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1



Caches: Overview

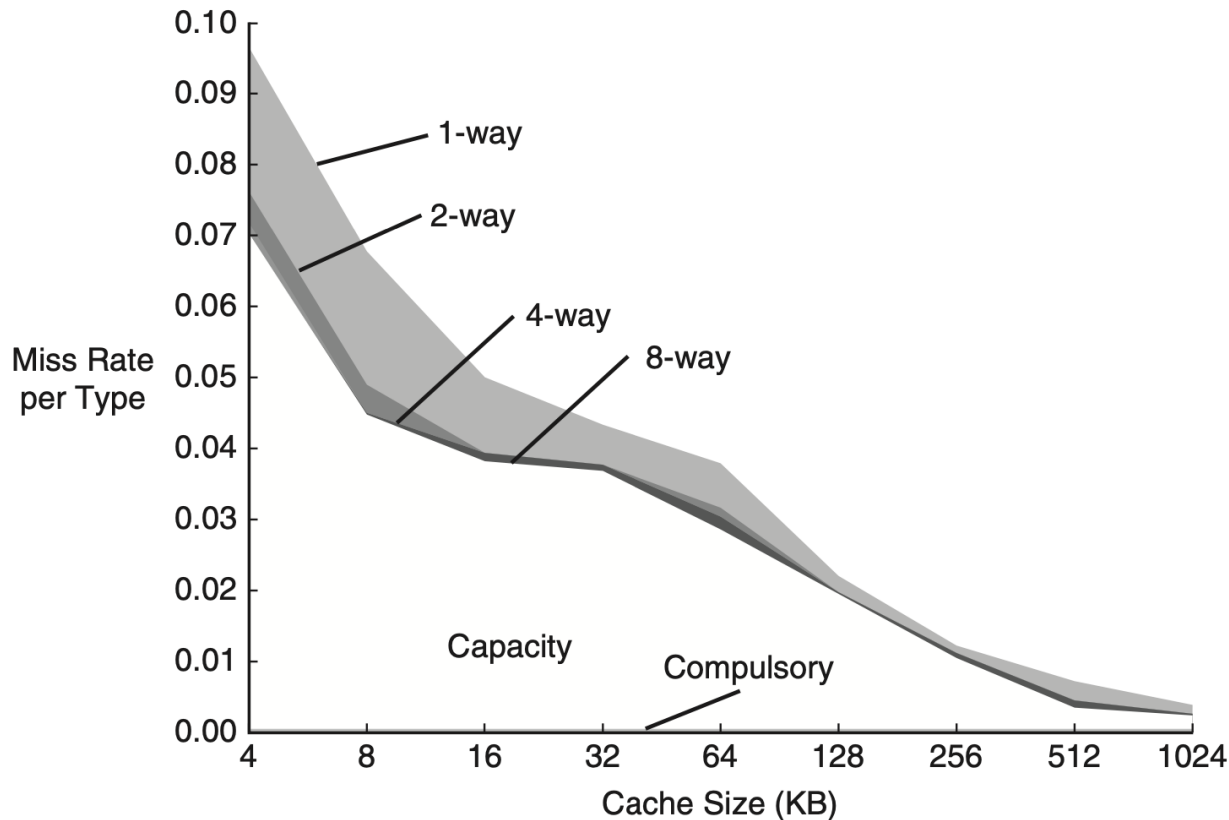


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark

Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.

- **Higher associativity:** generally lower miss rates

Caches: Recall from lecture

Types of cache miss

- **Cold (compulsory)** miss
 - Occurs on first access to a block
- **Conflict** miss
 - Most caches limit placement to small subset of available slots
 - e.g., block i must be placed in slot $(i \bmod 4)$
 - Cache may be large enough, but multiple lines map to same slot
 - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
- **Capacity** miss
 - Set of active cache blocks (working set) larger than cache
- **Coherency** miss
 - Multiprocessor systems: see later in the course

Caches: Recall from lecture

What to do on a write-hit?

- Write-through
 - Write immediately to memory
 - Memory is always consistent with the cache copy
 - Slow: what if the same value (or line!) is written several times
- Write-back
 - Defer write to memory until replacement of line
 - Need a **dirty** bit
 - \Rightarrow indicates line is different from memory
 - Higher performance (but more complex)

Caches: Recall from lecture

What to do on a write-miss?

- Write-allocate (load into cache, update line in cache)
 - Good if more writes to the location follow
 - More complex to implement
 - May evict an existing value
 - Common with write-back caches
- No-write-allocate (writes immediately to memory)
 - Simpler to implement
 - Slower code (bad if value subsequently re-read)
 - Seen with write-through caches

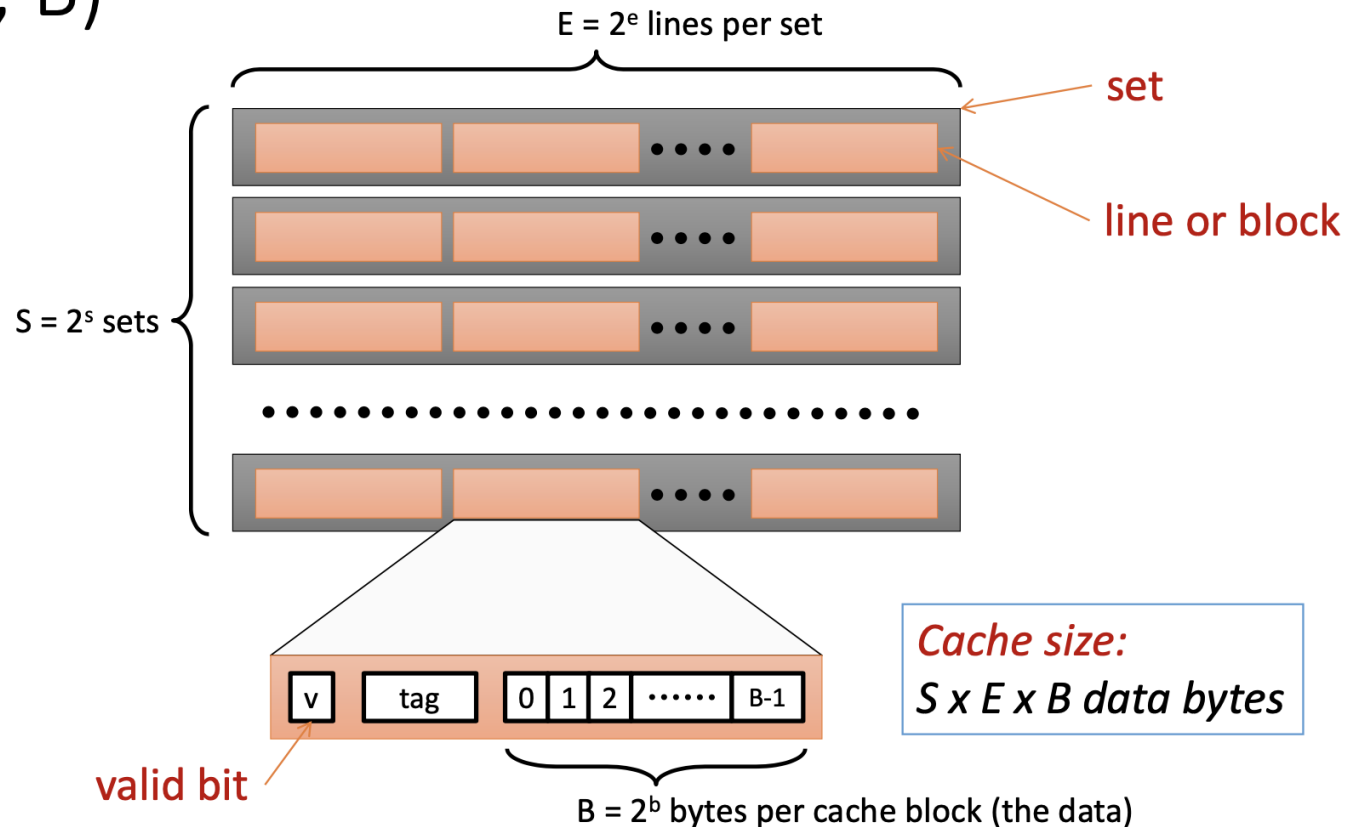
Caches: Recall from lecture

Other hardware cache features

- Unified
 - Serves both instruction and data fetches
- Private
 - Only one core uses this cache
- Shared
 - Multiple cores share the cache
- Inclusive
 - Anything in this cache is **also** in every lower-level cache
- Exclusive
 - Anything in this cache is **not** in any lower-level cache

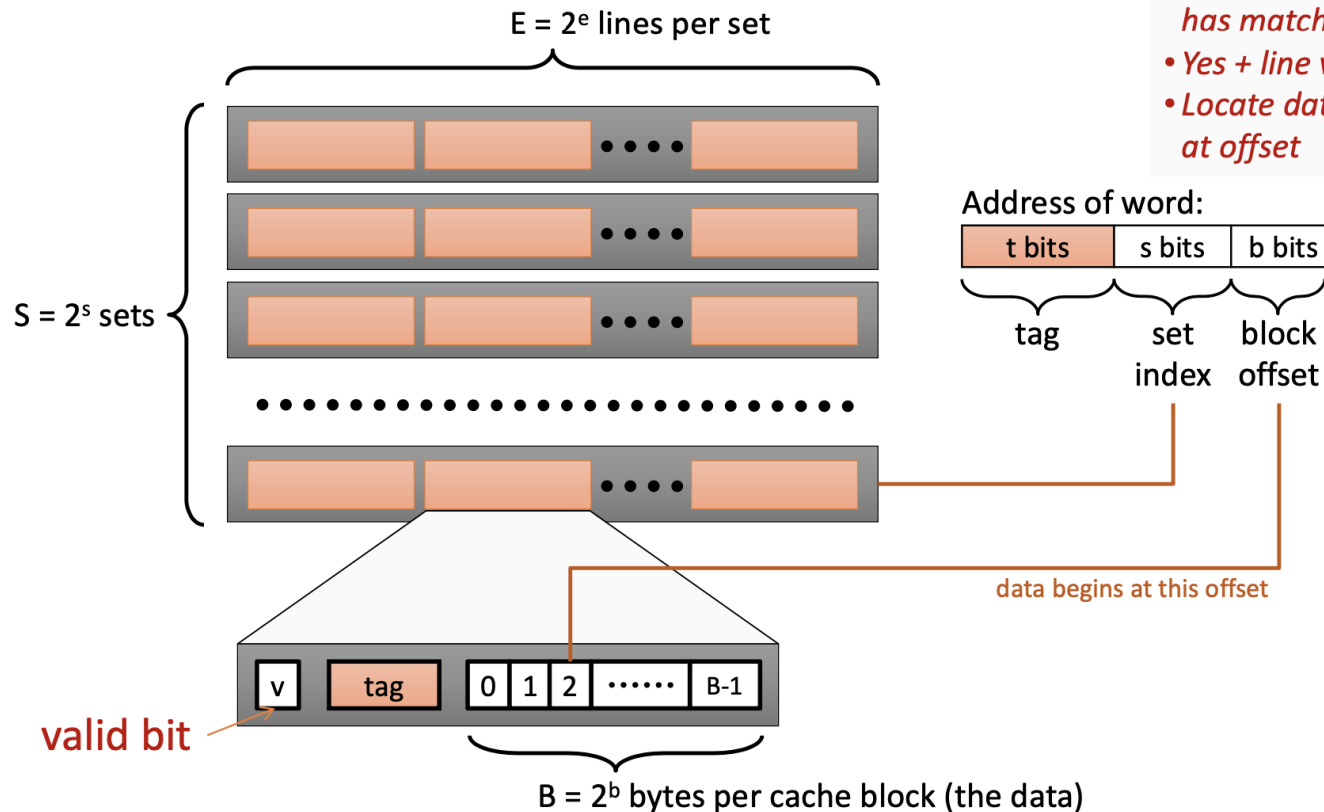
Caches: Recall from lecture

General cache organization (S, E, B)



Caches: Recall from lecture

Cache read



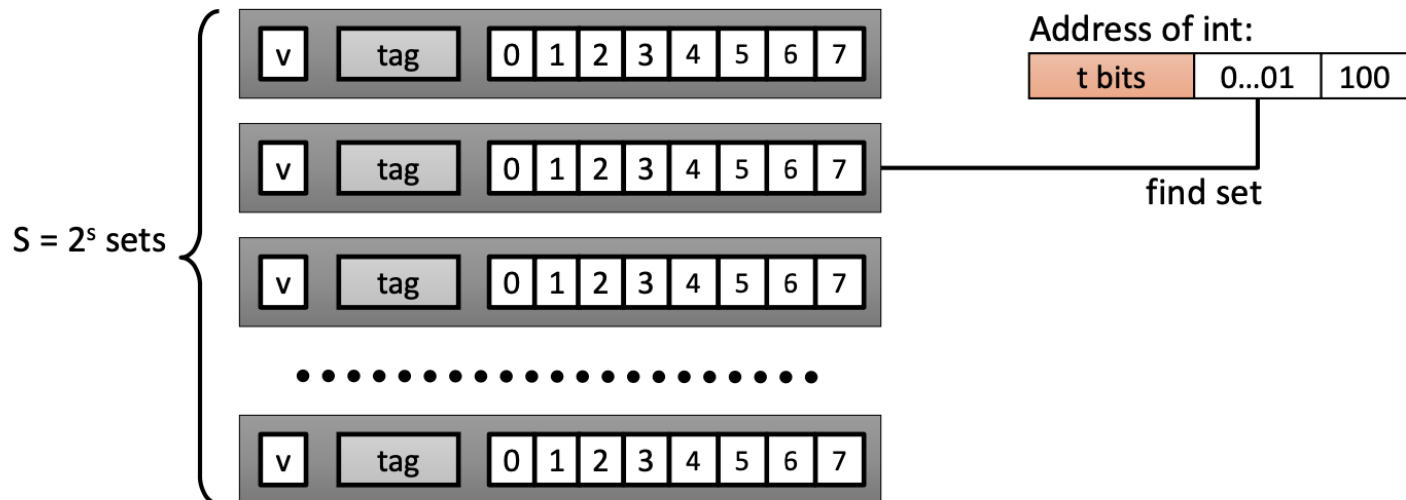
- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Caches: Recall from lecture

Direct mapped cache ($E = 1$)

Direct mapped: One line per set

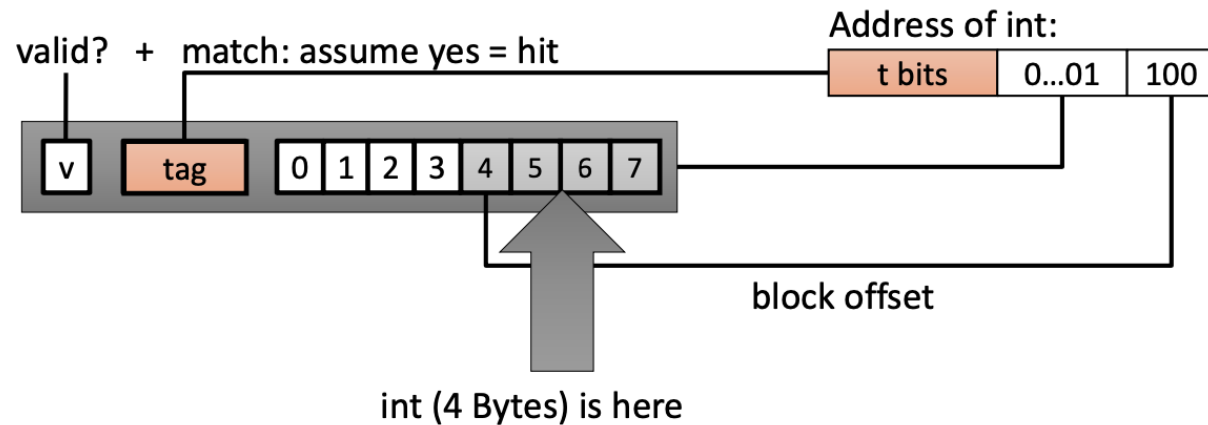
This example: cache block size 8 bytes



Caches: Recall from lecture

Direct mapped cache ($E = 1$)

Direct mapped: One line per set
 This example : cache block size 8 bytes



No match: old line is evicted and replaced

Virtual Memory

Virtual Memory

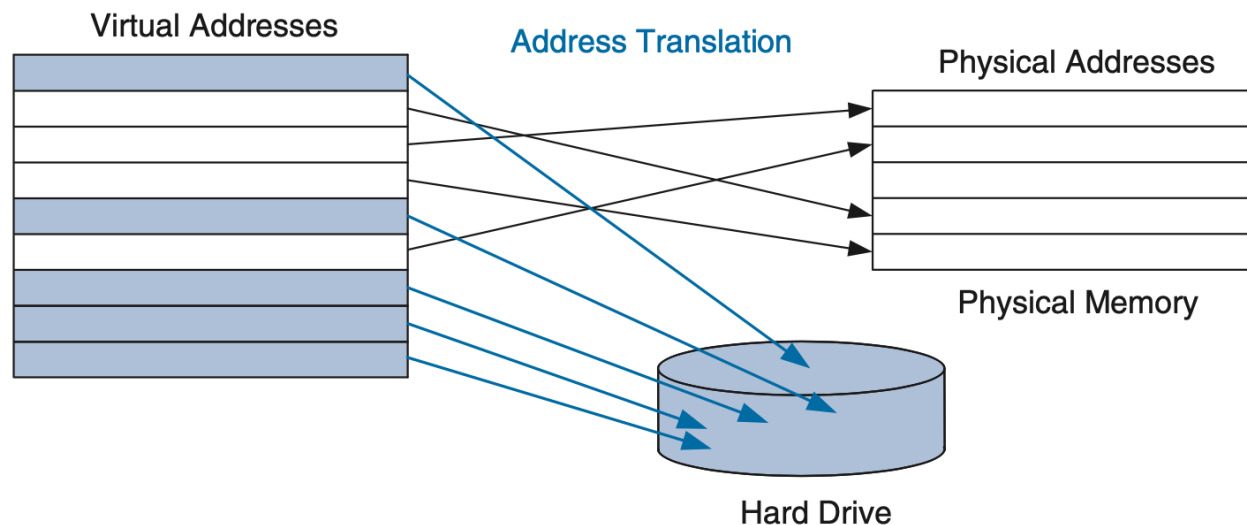
- Let us first look at how virtual memory works
- Then we can check out why it's a useful concept
- **Before** we go to virtual memory, how do virtual memory and caches relate?

Virtual Memory

- **Some Terminology**
- **Physical Memory** = Main Memory = DRAM, often 8, 16, 32GBs in modern systems
- **Virtual Memory** = Disk / Hard Drive, ranges from 120-1000GB
- Programs can access data **anywhere in virtual memory**: so they must use **virtual addresses** that specify location in virtual memory
- Physical memory holds a **subset of most recently accessed virtual memory**: physical memory acts as a cache for virtual memory

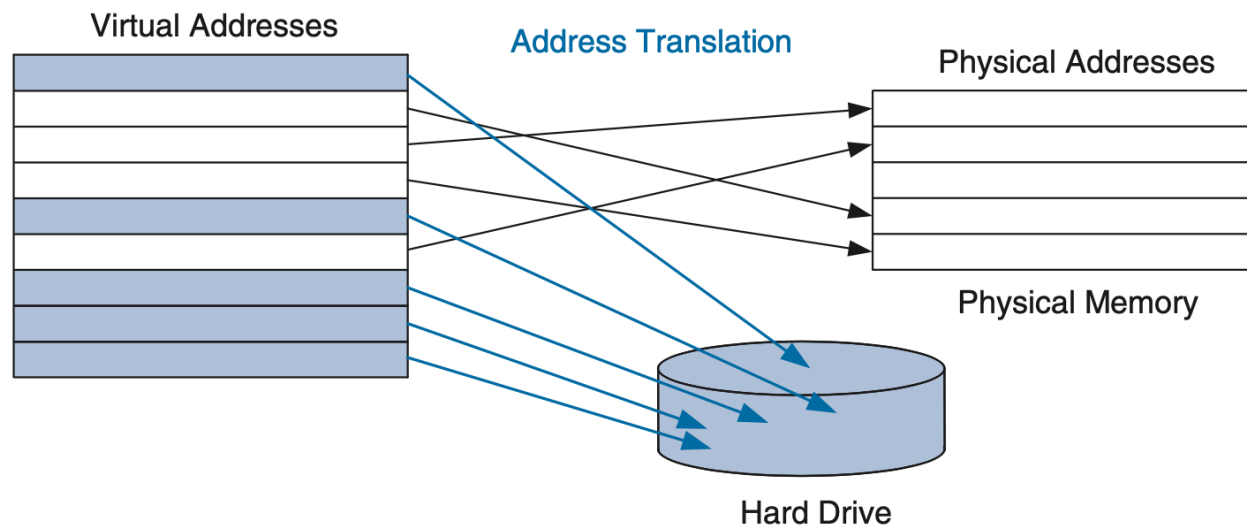
Virtual Memory

- **Virtual Memory:** divided into **virtual pages** (typically 4KB size)
- **Physical Memory:** divided into **physical pages** (same size)
- **Virtual page** may be located in i) **physical memory** (DRAM) or on **hard drive** (disk)



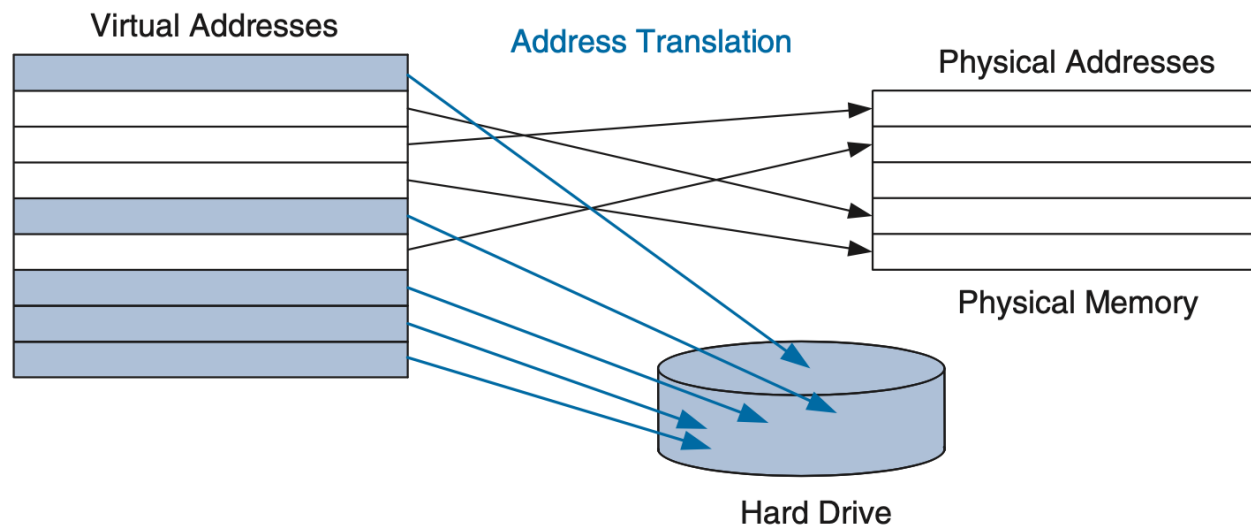
Virtual Memory

- **Address translation:** Process of determining physical address, given a virtual address
- **Page fault:** if processor tries to attempt to access a virtual address that is not in physical memory (see exception slides)



Virtual Memory

- **Page Table:** contains entry for each **virtual page**, indicating whether its in physical memory or on disk
- **Each load store:** Requires page table access, followed by access of physical memory (page table **also in physical memory**, so effectively 2x Physical memory access: TLB)



Virtual Memory: Address Translation

Virtual Memory

- **2GB**= 2^{31} -byte virtual memory
- **128MB**= 2^{27} -byte physical memory
- **4KB**= 2^{12} -byte pages
- $2^{31} / 2^{12} = 2^{19}$ virtual pages (19 bit VPN)
- $2^{27} / 2^{12} = 2^{15}$ physical pages (15 bit PPN)
- Physical memory can hold 1/16 of virtual pages at a time

Physical
Page
Number

7FFF
7FFE
⋮
0001
0000

Physical Addresses

0x7FFF000 - 0x7FFFFFFF
0x7FFE000 - 0x7FFEFFFF
⋮
0x0001000 - 0x0001FFF
0x0000000 - 0x0000FFF

Physical Memory

Virtual Addresses

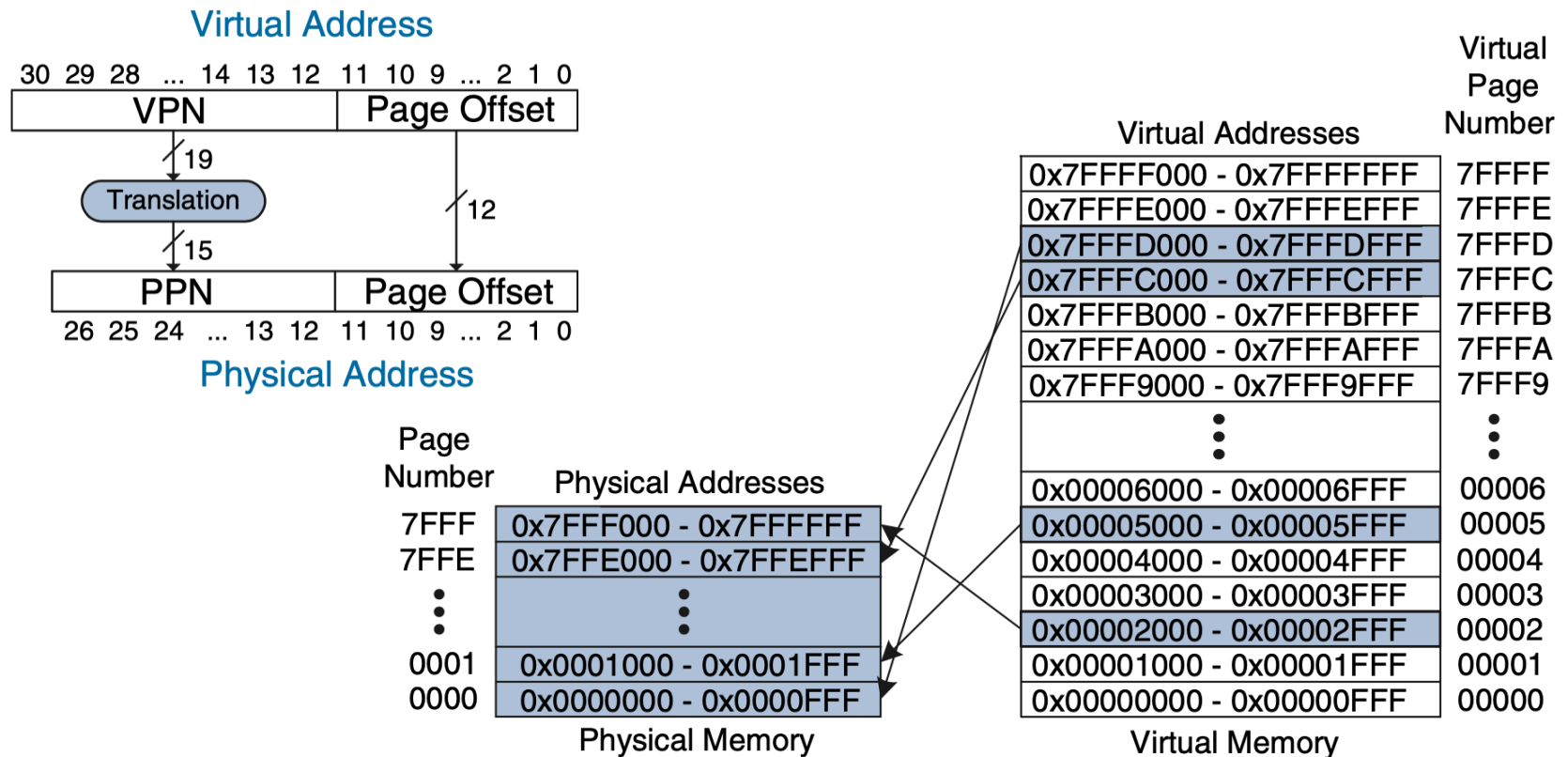
0x7FFFF000 - 0x7FFFFFFF	7FFFF
0x7FFFE000 - 0x7FFFEFFF	7FFFE
0x7FFFD000 - 0x7FFFDFFF	7FFFD
0x7FFFC000 - 0x7FFFCFFF	7FFFC
0x7FFFB000 - 0x7FFFBFFF	7FFFB
0x7FFFA000 - 0x7FFFAFFF	7FFFA
0x7FFF9000 - 0x7FFF9FFF	7FFF9
⋮	⋮
0x00006000 - 0x00006FFF	00006
0x00005000 - 0x00005FFF	00005
0x00004000 - 0x00004FFF	00004
0x00003000 - 0x00003FFF	00003
0x00002000 - 0x00002FFF	00002
0x00001000 - 0x00001FFF	00001
0x00000000 - 0x00000FFF	00000

Virtual
Page
Number

Virtual Memory

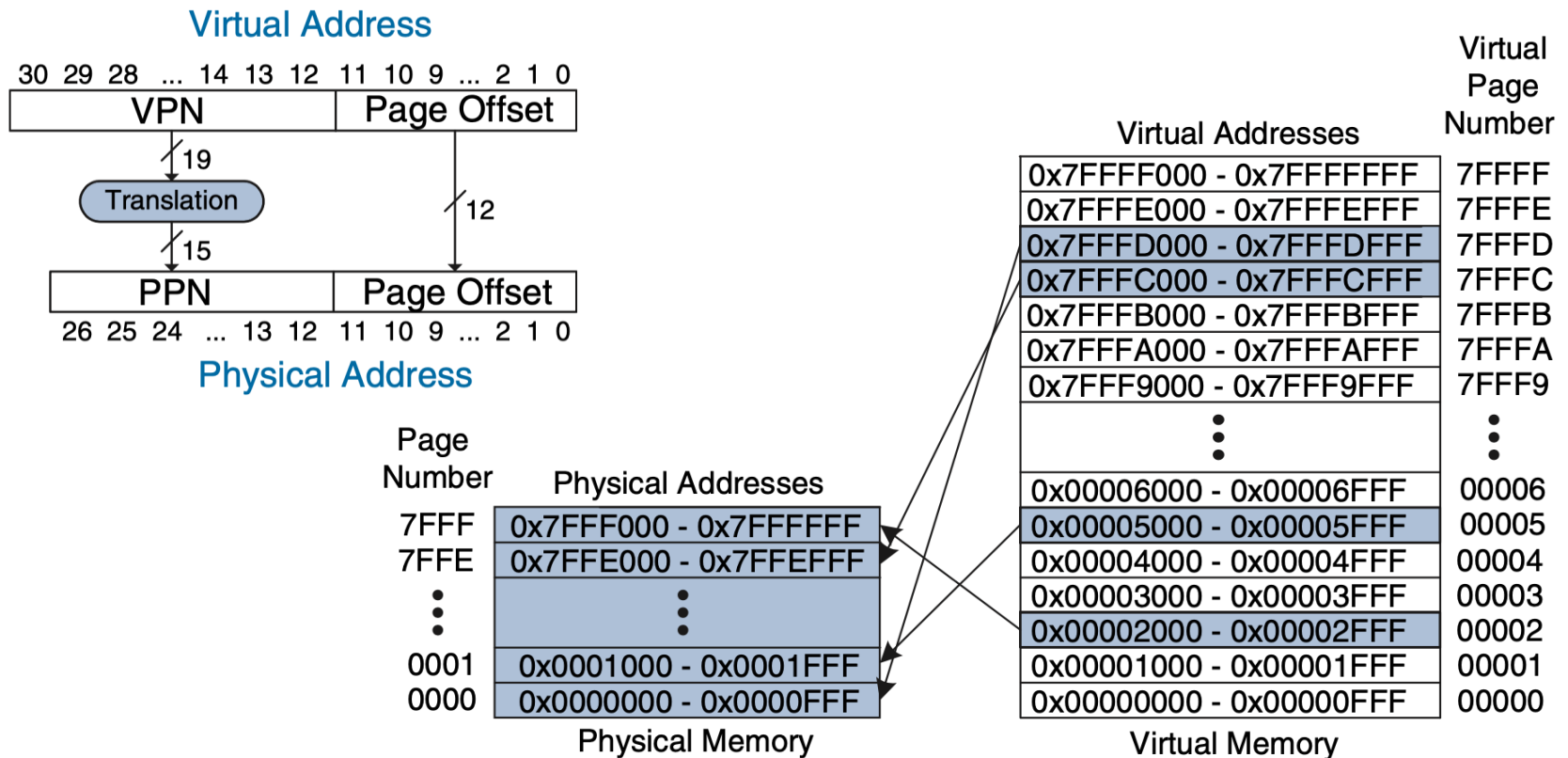
Virtual Memory

- Task: Physical address of virtual address 0x247C using virtual memory system?



Virtual Memory

- **Physical address of virtual address 0x247C** using virtual memory system
- 12bit page offset (0x47C) needs no translation. Remaining 19 bits are the VPN, so virtual address 0x247C found in virtual page 0x2: PPN: 0x7FFF, Physical address: 0x7fff47C



Virtual Memory

- **Issue?** What we did by hand is tedious and time consuming, also the processor needs a **general way** and need to store millions of mappings for multiple processes
- **Solution:** store mappings VPN->PPN in a **table**, the **page table**

Virtual Memory

- Processor uses **page table** to translate VPN->PPN
- Contains **entry** for each virtual page:
Valid bit (if currently in physical memory)
- Indexed with **virtual page number**
- **Entry 5**: specifies virtual page 5 maps to physical page 1
- **Entry 6**: Invalid (V=0) so located on disk

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

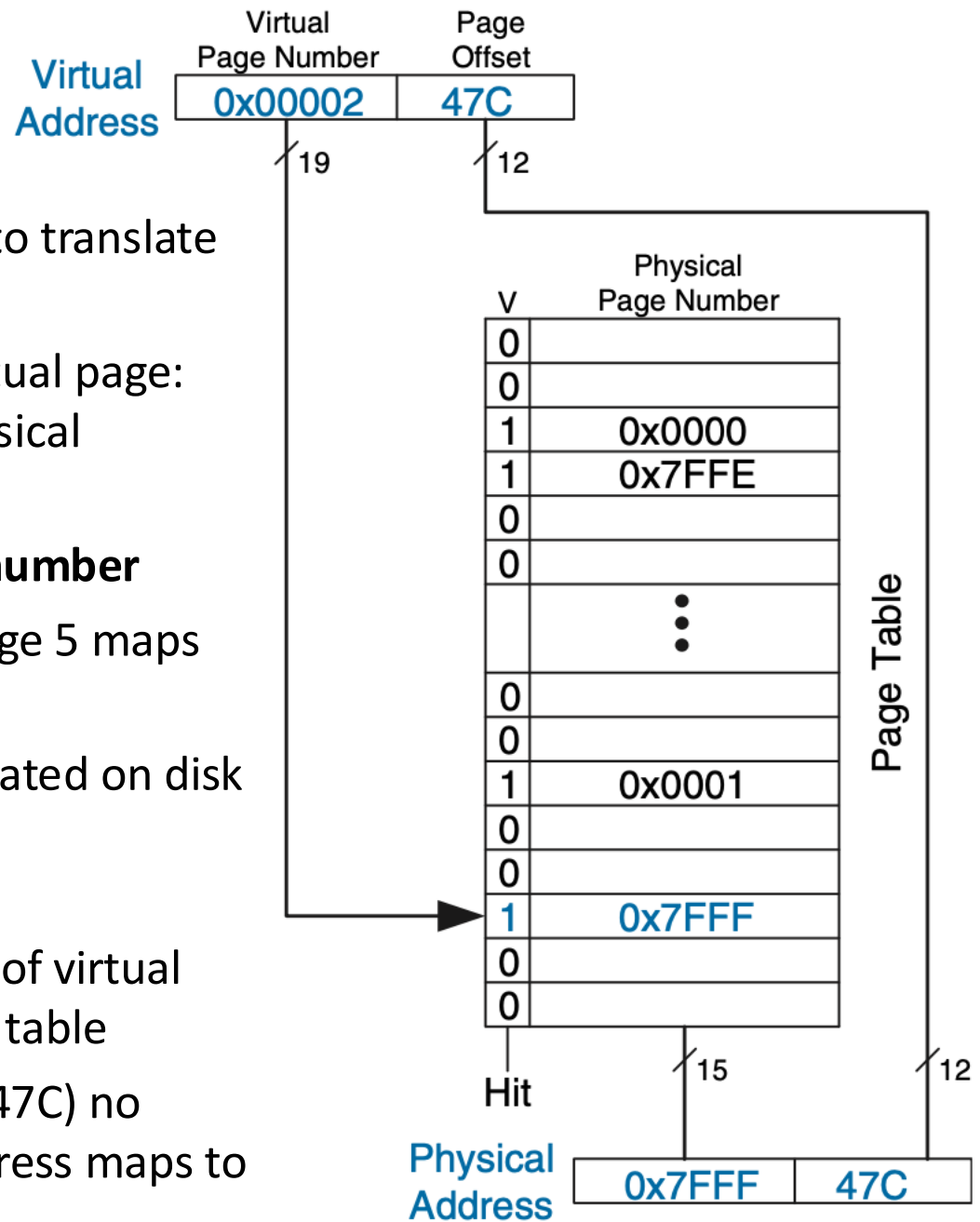
Virtual Memory

- Processor uses **page table** to translate VPN->PPN
- Contains **entry** for each virtual page:
Valid bit (if currently in physical memory)
- Indexed with **virtual page number**
- **Entry 5**: specifies virtual page 5 maps to physical page 1
- **Entry 6**: Invalid (V=0) so located on disk
- **Task**: Find physical address of virtual address 0x247C using page table

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

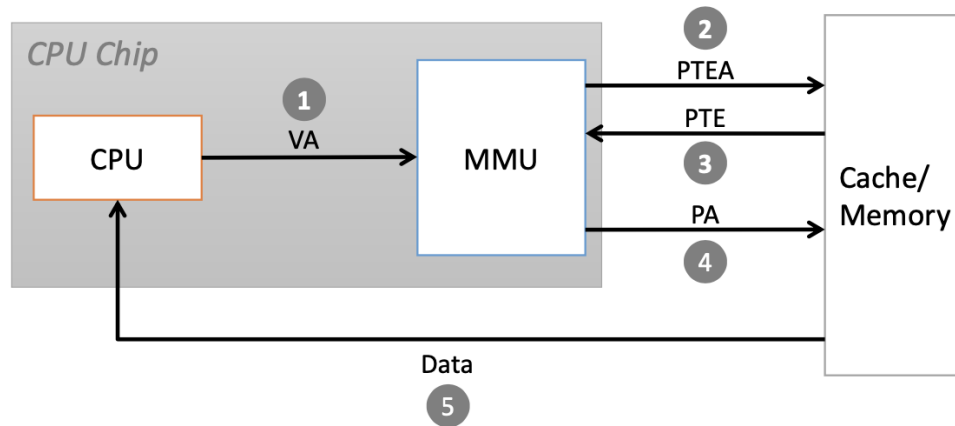
Virtual



- Processor uses **page table** to translate VPN->PPN
- Contains **entry** for each virtual page:
Valid bit (if currently in physical memory)
- Indexed with **virtual page number**
- **Entry 5:** specifies virtual page 5 maps to physical page 1
- **Entry 6:** Invalid (V=0) so located on disk
- **Task:** Find physical address of virtual address 0x247C using page table
- **Sol:** 12 bit page offset (0x247C) no translation; 0x2 virtual address maps to 0x7FFF, in total: 0x7FFF47C

Virtual Memory

- If we were to access Entry5: page hit



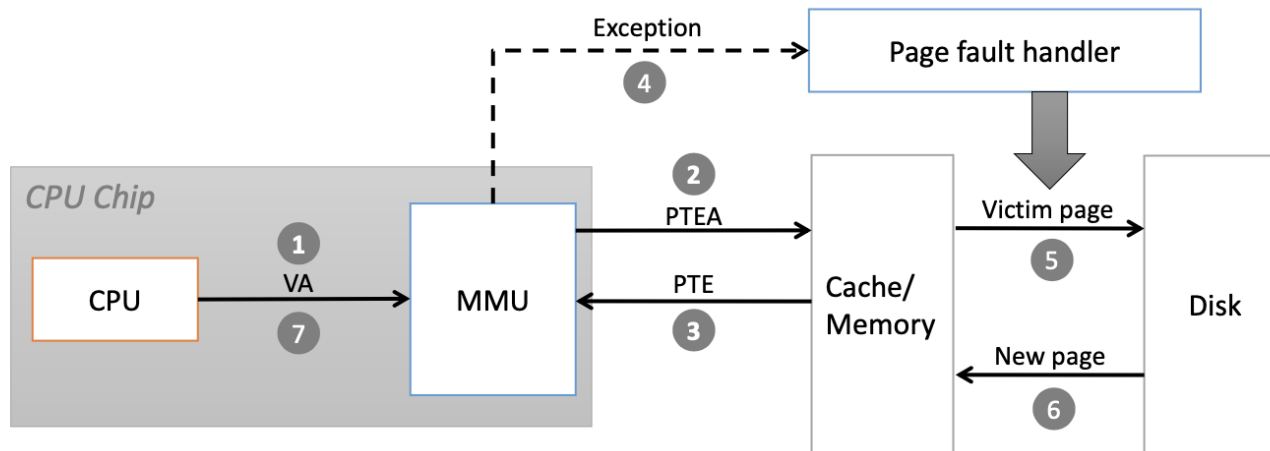
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Virtual Memory

- If we were to access Entry6: page fault (V=0)



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim page to evict (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

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Virtual Memory

- **Page table:** can be stored anywhere in physical memory at discretion of the OS
- **Processor** uses dedicated register, called **page table register** to store base address and page table in **physical memory**

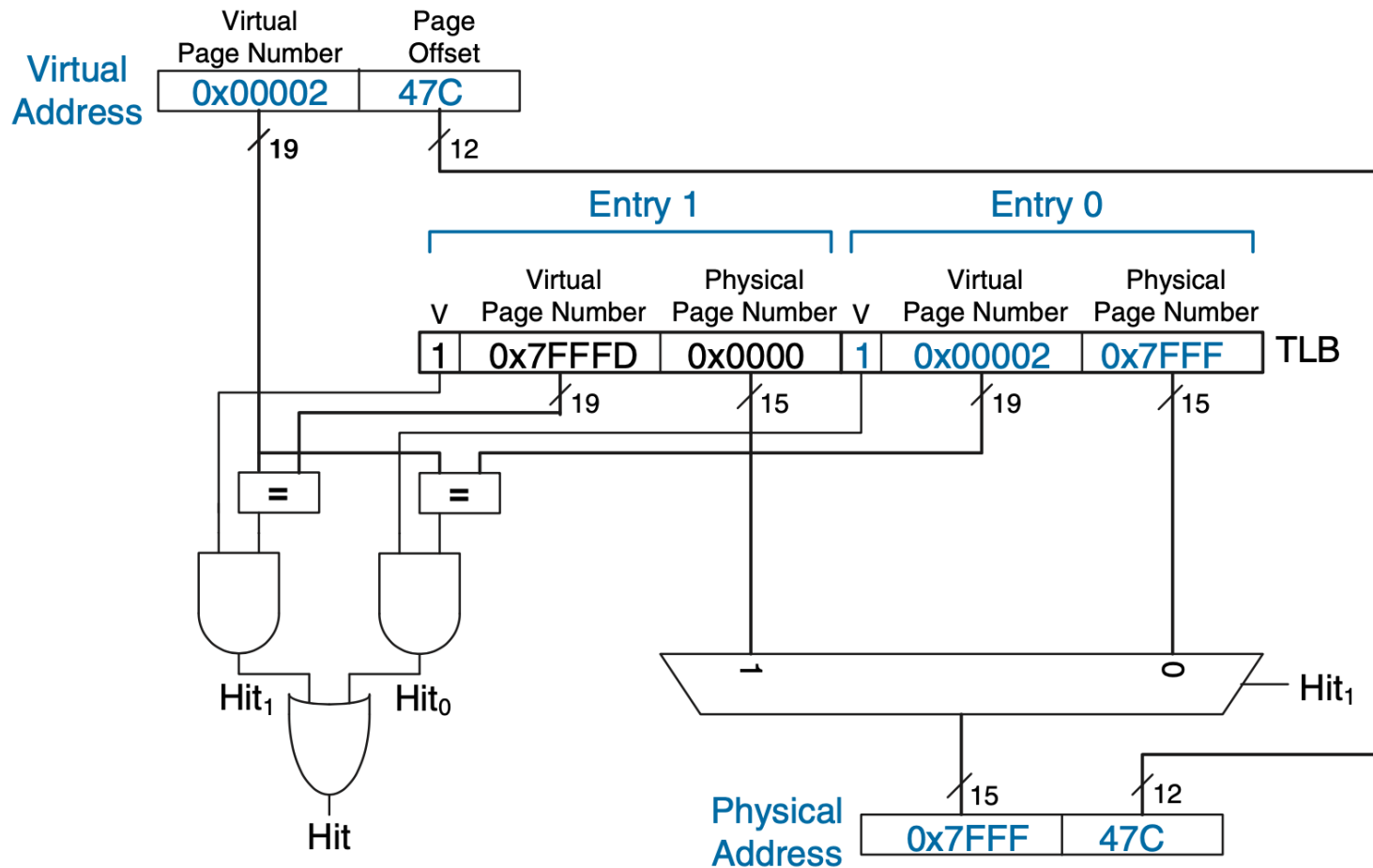
Virtual Memory: Translation Lookaside Buffer (TLB)

Virtual Memory

- **Virtual Memory:** would have severe performance impact if we needed a page table read on every load/store (2x physical memory access)
- **Idea:** page table accesses have great spatial & temporal locality & **large page size** => lets **cache PTEs**
- **Processor keeps** last several page table entries in small cache called “translation lookaside buffer” (TLB)
- **Processor** “looks aside” to find translation in TLB before having to access page table in physical memory
- **TLBs** have 16-512 entries (quite small): though TLBs have hit rate > 99%

Virtual Memory

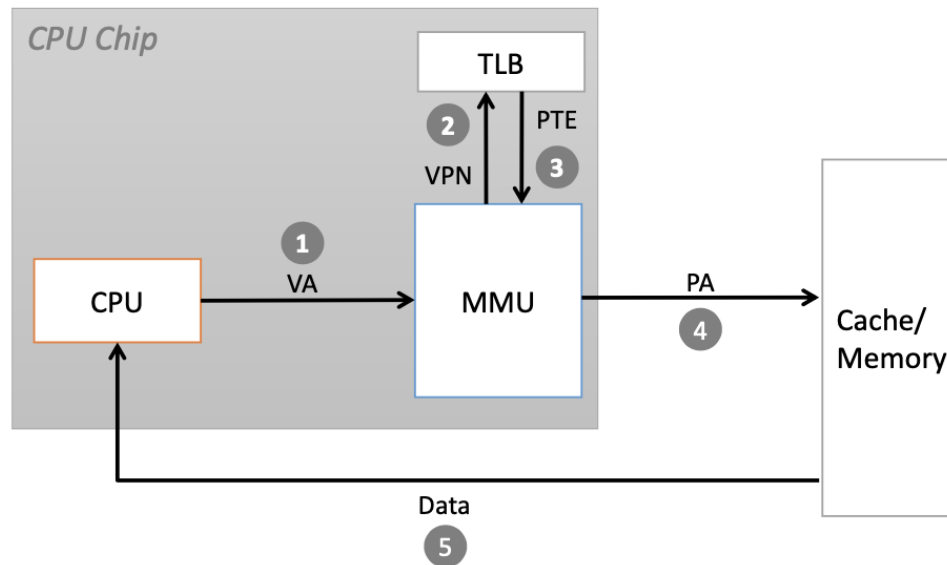
- Accessing VPN 0x2 hits in this two entry TLB



Virtual Memory

- **Accessing** VPN 0x2 hits in this two entry TLB

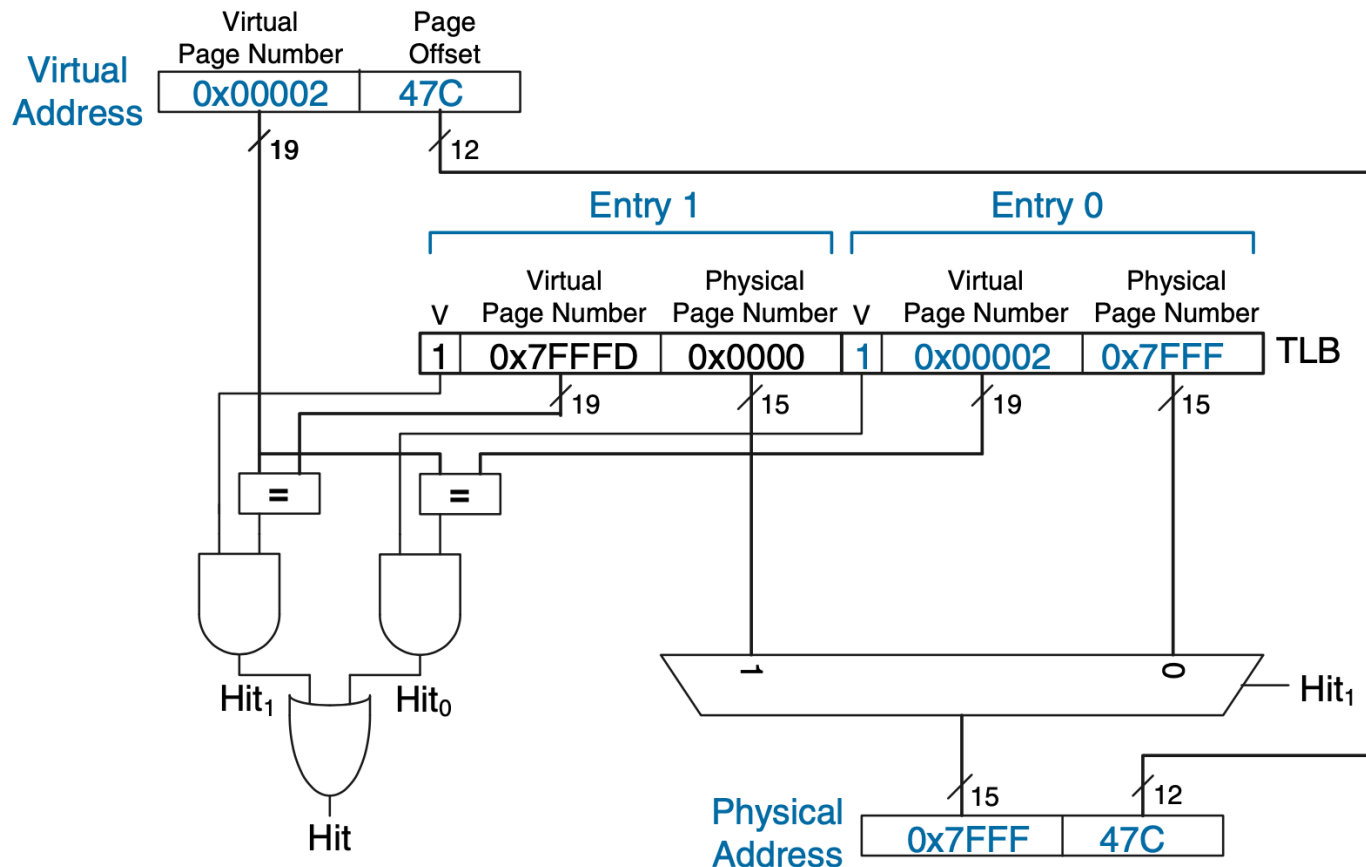
TLB hit



A TLB hit eliminates a memory access

Virtual Memory

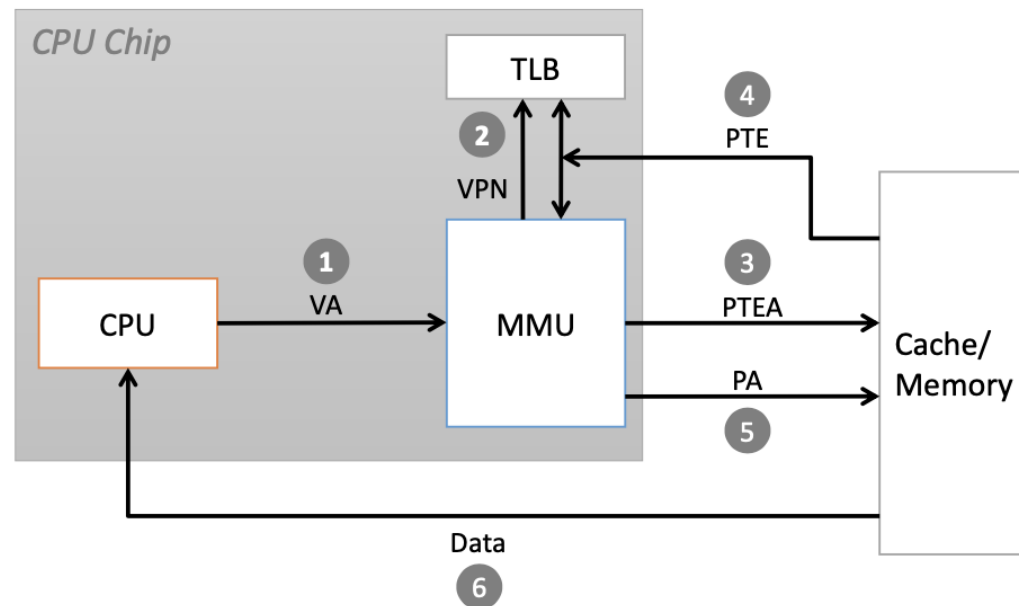
- **Accessing** VPN 0x5FB0 misses in TLB: need to access page table in physical memory



Virtual Memory

- **Accessing** VPN 0x5FB0 misses in TLB: need to access page table in physical memory

TLB miss



A TLB miss incurs an additional memory access (the PTE)

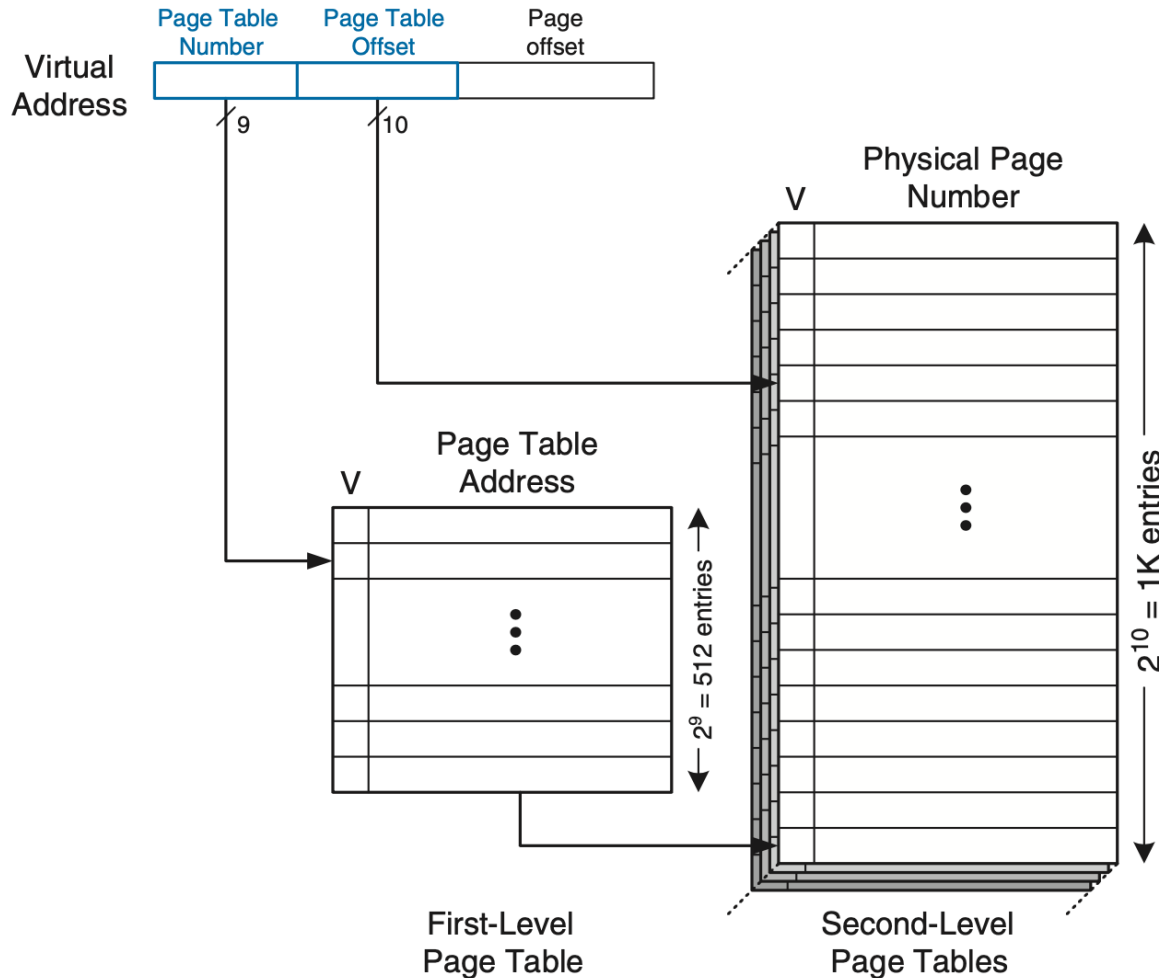
Fortunately, TLB misses are rare (we hope)

Virtual Memory: Multi-Level- Pagetable

Virtual Memory

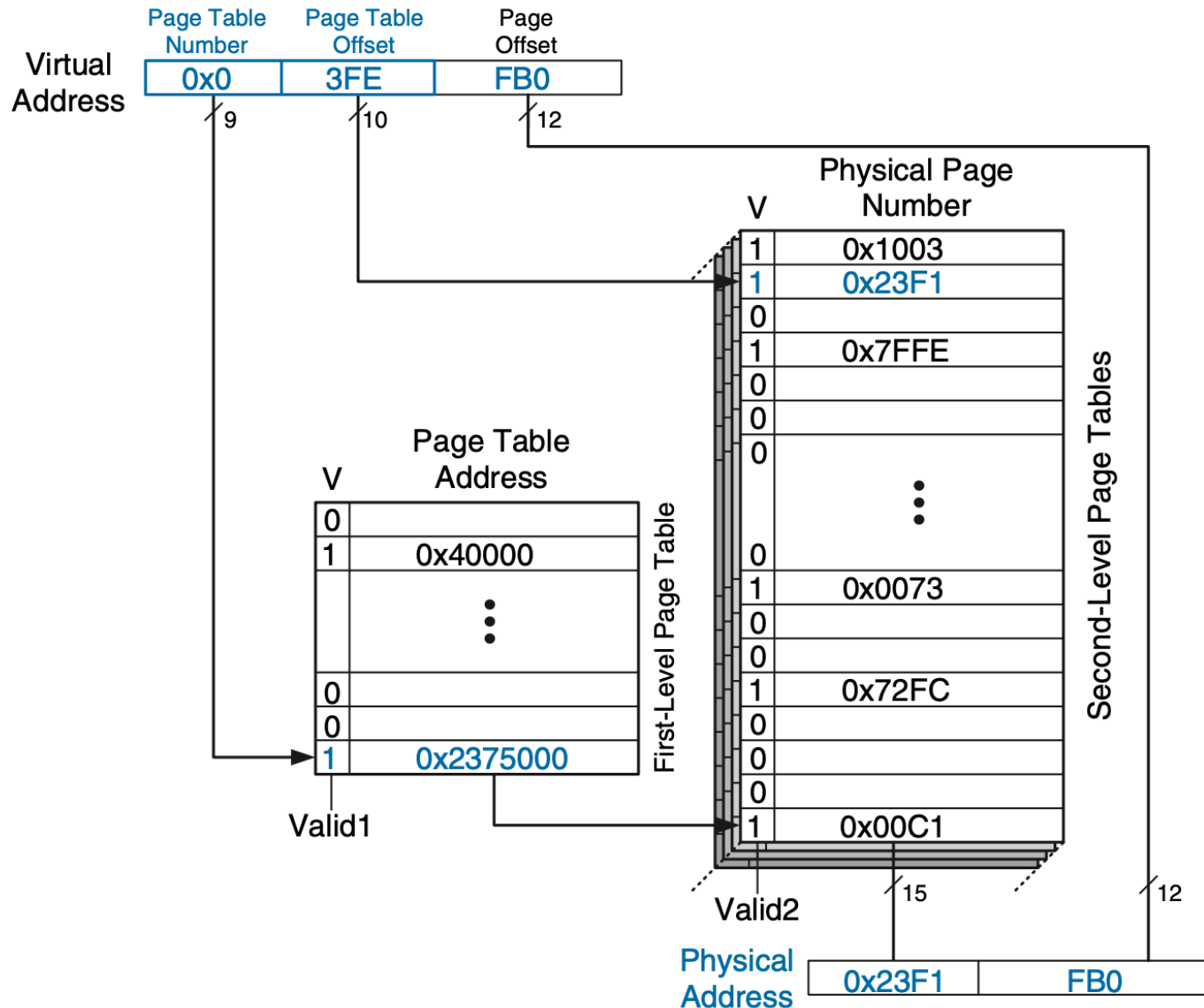
- **In lecture:** its not feasible to have an entire page table (not even for one process)
- **To conserve memory:** page tables can be broken up into multiple levels: first level page table always kept in physical memory, indicates where second level page tables are stored in **virtual memory**
- **In a 2 level page table:** 2nd page table contains actual physical addresses

Virtual Memory



- **Page Table Number:** indexes 1st level page table (gives base address of second table)
- **Page Table Offset:** indexes 2nd level page table

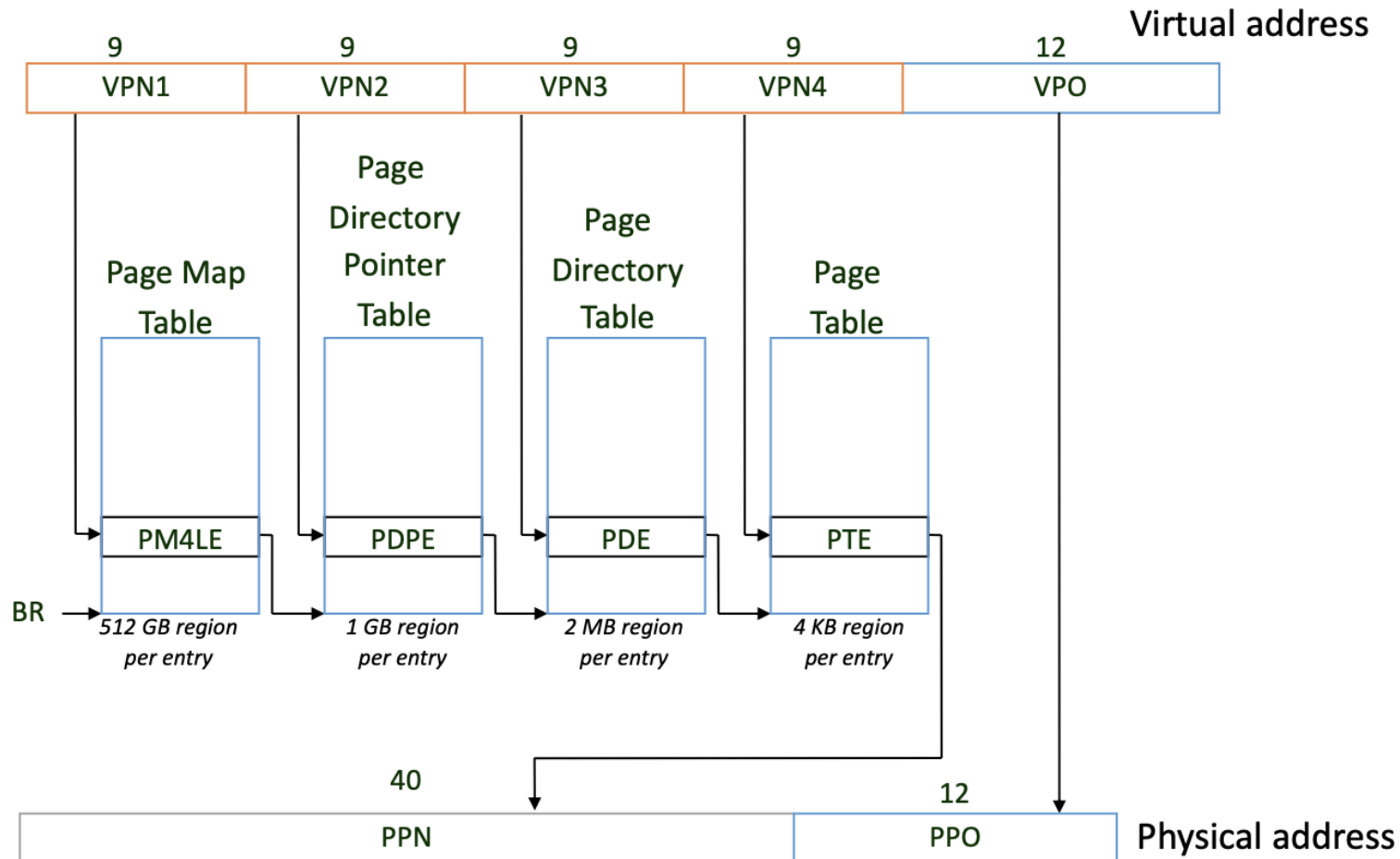
Virtual Memory



- **Example:**
accessing
virtual address
0x003FEFB0
- Only VPN
needs
translation:
- Page Table
number: 0x0
- Page Table
Offset 0x3FE
- PPN:
0x23F1FB0

Virtual Memory

- This concept generalizes to arbitrary levels



Why is Virtual Memory Useful?

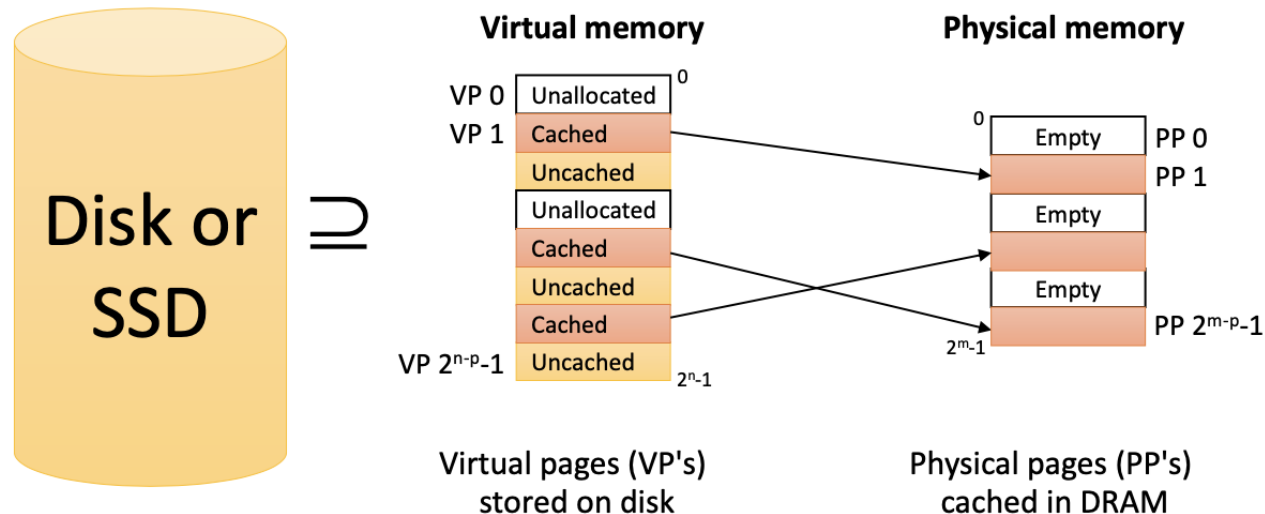
Virtual Memory

- **Efficient use** of limited main memory (RAM)
 - Use RAM as a cache for the parts of a virtual address space
 - some non-cached parts stored on disk
 - some (unallocated) non-cached parts stored nowhere
 - Keep only active areas of virtual address space in memory
 - transfer data back and forth as needed
- **Simplifies** memory management for programmers
 - Each process gets the same full, private linear address space
- **Isolates** address spaces
 - One process can't interfere with another's memory
 - because they operate in different address spaces
 - User process cannot access privileged information
 - different sections of address spaces have different permissions

Virtual Memory

1: VM as a tool for caching

- Virtual memory: array of $N = 2^n$ contiguous bytes
 - think of the array (allocated part) as being stored on disk
- Physical main memory (DRAM) = cache for allocated virtual memory
- Blocks are called pages; size = 2^p



Virtual Memory

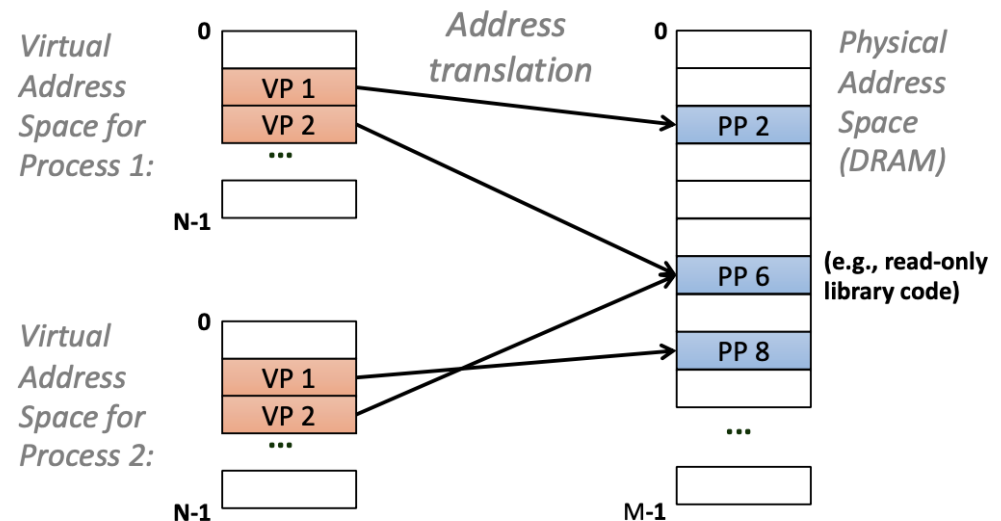
- Remember **shared object files** from linking?

2. VM as a tool for memory management

Key idea: each process has its own virtual address space

→ each process needs its own page table!

- Allocated physical pages are scattered in memory
 - Well-chosen virtual→physical mappings simplify memory allocation and management

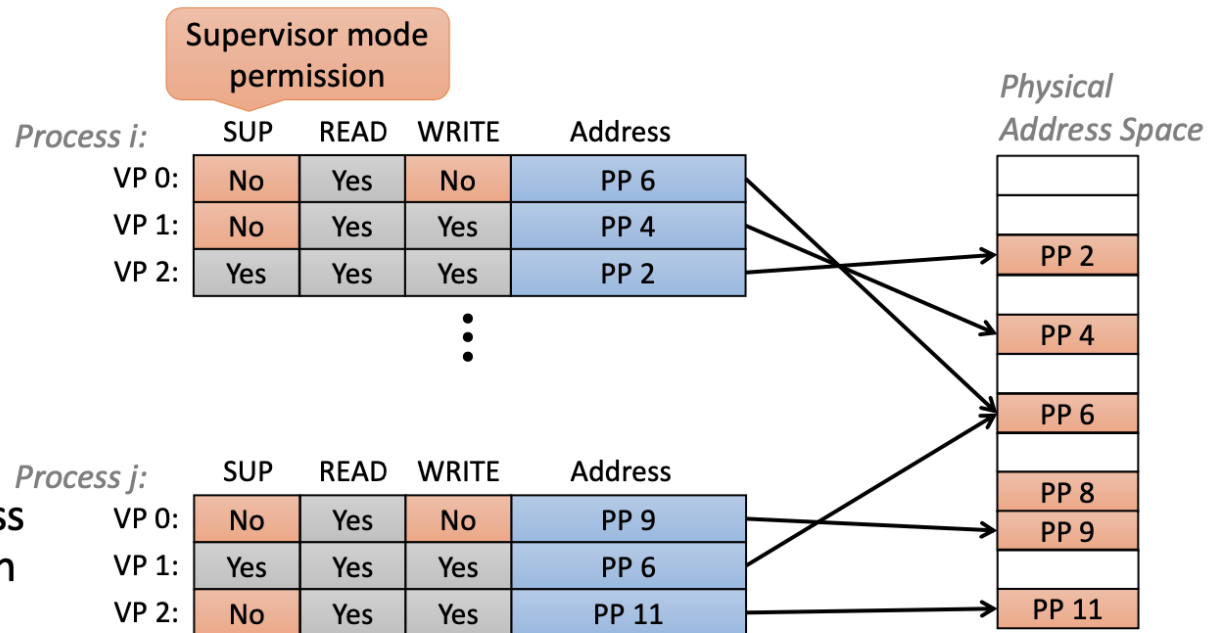


Virtual Memory

- Remember **making stack not executable** from attacks?

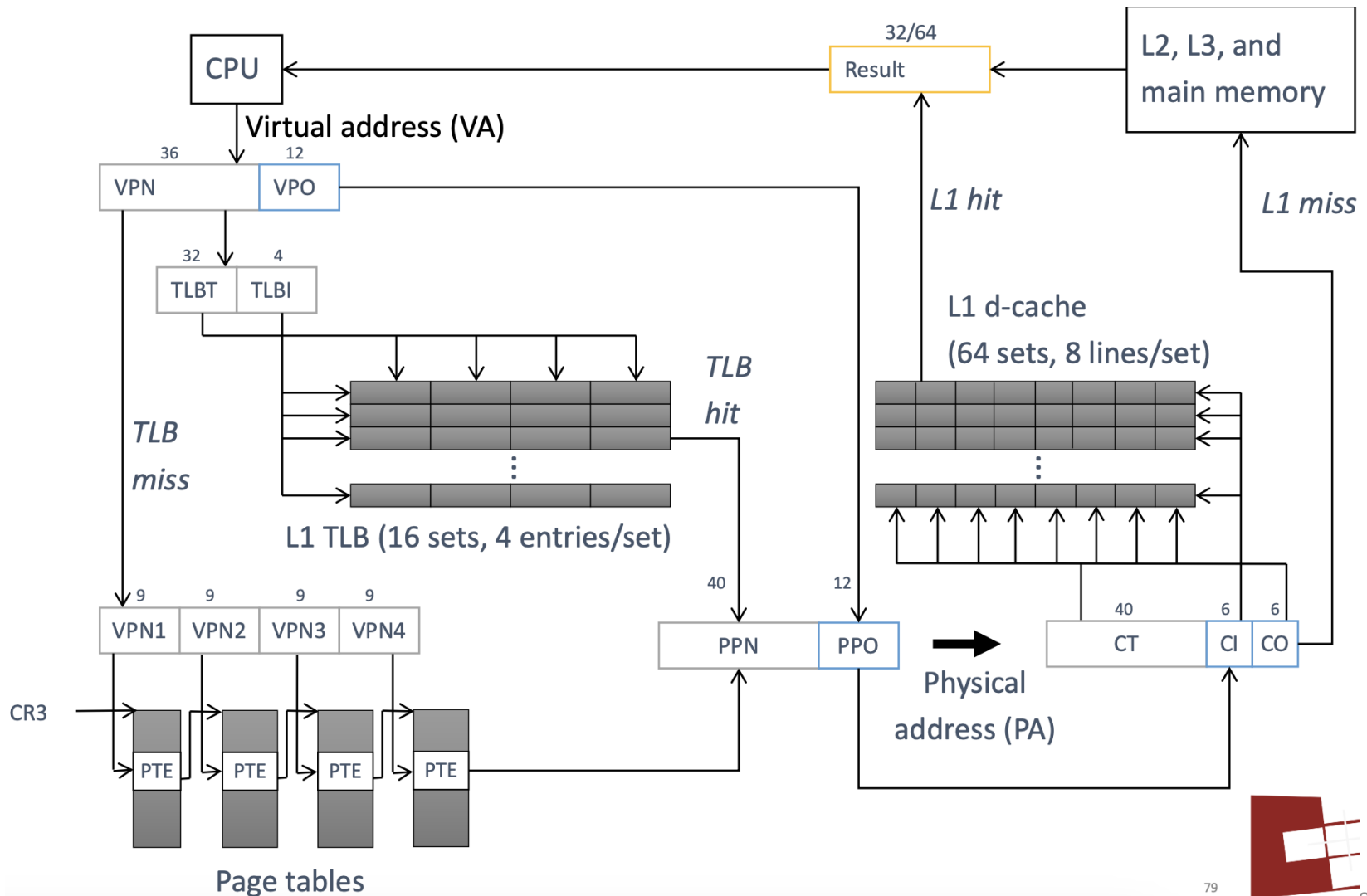
3. VM as a tool for memory protection

- Extend Page Table Entries (PTEs) with permission bits
- Page fault handler checks these before remapping
 - If violated, send process SIGSEGV (segmentation fault)



Big Picture: Relation between Caches and Virtual Memory

Caches and Virtual Memory



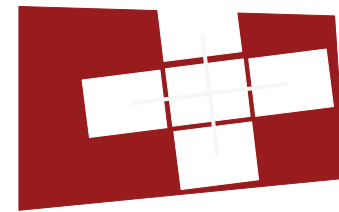
Virtual Memory

- “Virtually/Physically **Indexed**” - “Virtually/Physically **Tagged**” depends on how we **access the cache**
- Virtually Indexed – Virtually Tagged
- Virtually Indexed – Physically Tagged
- Physically Indexed – Virtually Tagged
- Physically Indexed – Physically Tagged

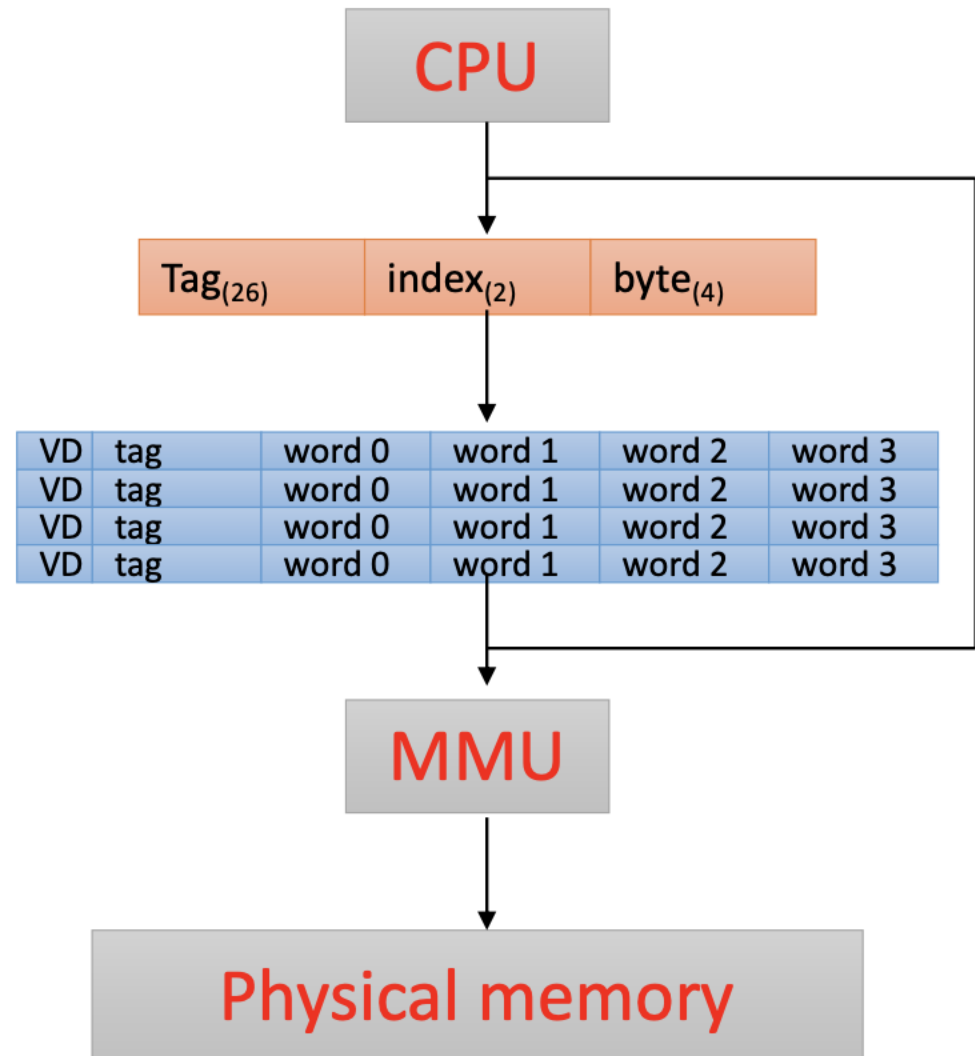
Virtual Memory

- “Virtually/Physically **Indexed**” - “Virtually/Physically **Tagged**” depends on how we **access the cache**
- **Virtually Indexed – Virtually Tagged**
- Virtually Indexed – Physically Tagged
- Physically Indexed – Virtually Tagged
- Physically Indexed – Physically Tagged

Virtually Indexed – Virtually Tagged



- **Only uses virtual address**
- **Homonyms:** same VAs -> different PAs
 - **Solution:** ASID (per process), Flush on context switch
- **Synonyms:** different VA -> same PA
 - **Solution:** Make read-only

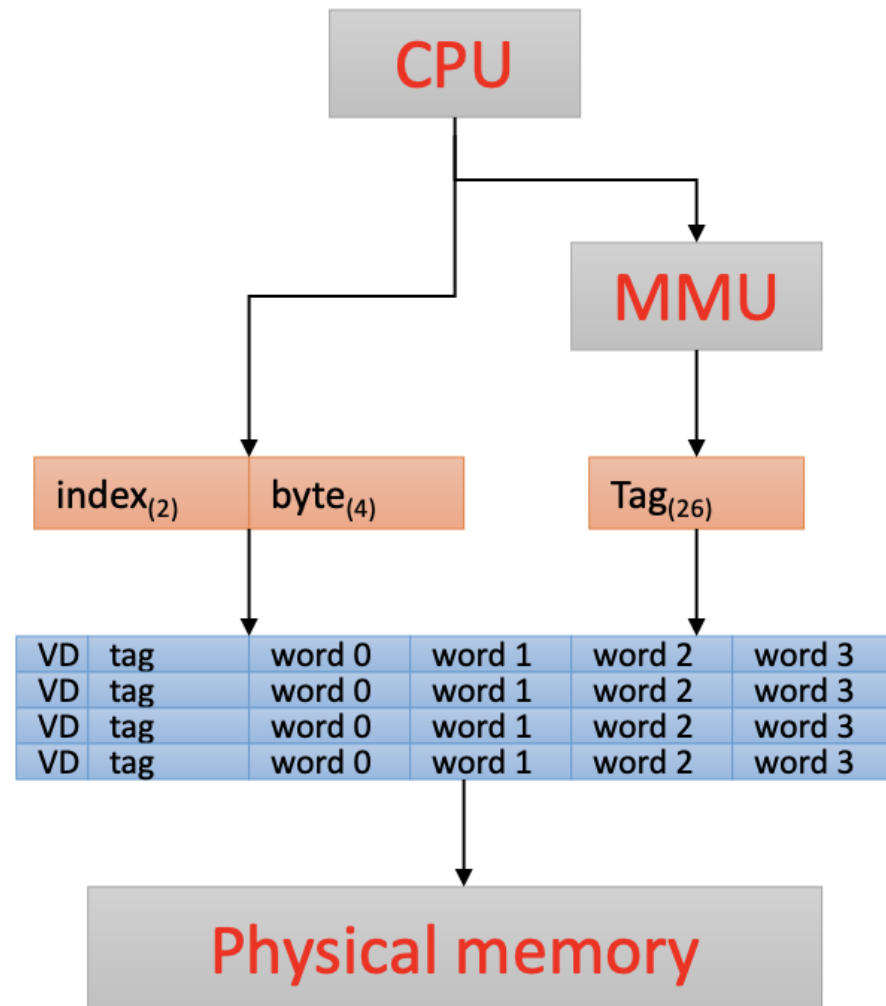


Virtual Memory

- “Virtually/Physically **Indexed**” - “Virtually/Physically **Tagged**” depends on how we **access the cache**
- Virtually Indexed – Virtually Tagged
- **Virtually Indexed – Physically Tagged**
- Physically Indexed – Virtually Tagged
- Physically Indexed – Physically Tagged

Virtually Indexed – Physically Tagged

- **Best of both worlds**
- **Virtually Indexed:** fast cache indexing (don't need to wait for translation)
- **Physically tagged:** no homonyms and synonym issues
- **Aliasing issue:** if cache is too big

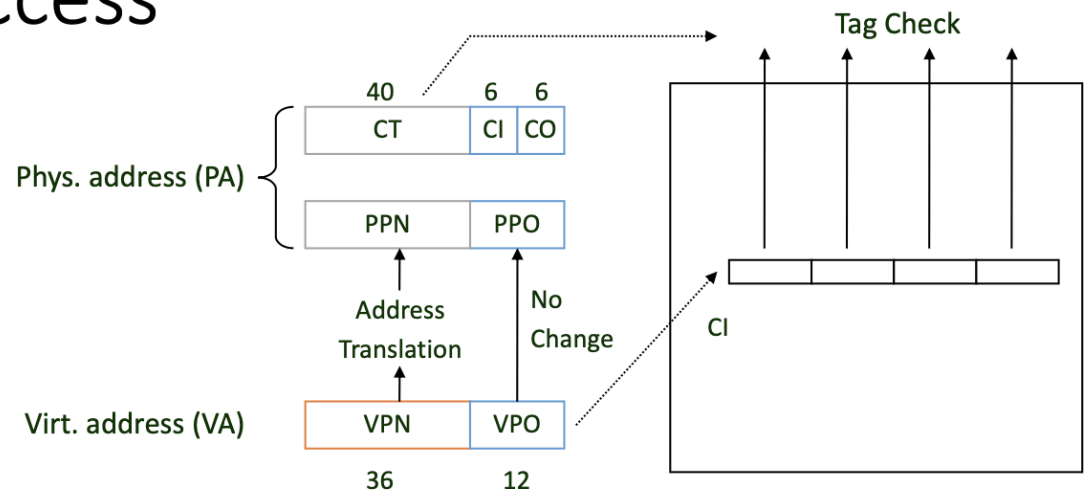


Virtually Indexed – Physically Tagged

- **Aliasing**
 - If two virtual addresses have the same physical address: as we **index virtually**, they get indexed to **different locations in the cache**
 - **This will lead to having two copies of the data block:** when these locations are update we get inconsistencies
- **Solution** (among others): **Reduce cache size**, i.e. s.t. VPO and PPO are the same: then the two virtual addresses will have the same **page offset**, so are **mapped to the same index**
- **What does this entail?**

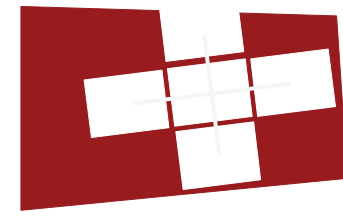
Virtually Indexed – Physically Tagged

Speeding up L1 access



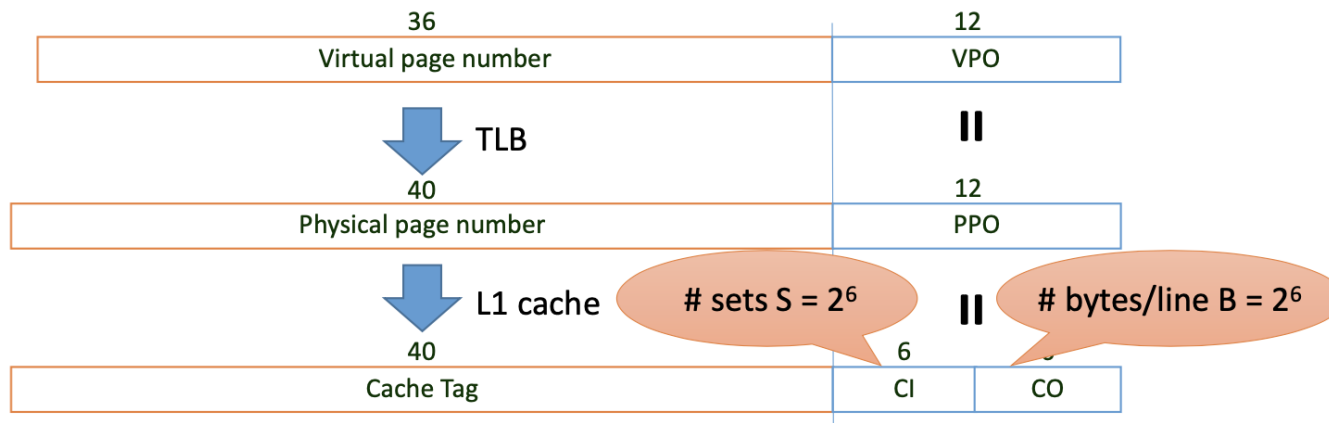
- Bits that determine CI *identical* in virtual and physical address
- Can index into cache *while* address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- **Cache carefully sized to make this possible**





Virtually Indexed – Physically Tagged

Why cache size isn't increasing over the years



- $\log_2(\text{cache size}) = \text{bits}(\text{CI}) + \text{bits}(\text{CO}) + \log_2(\text{associativity})$
- For Core i7: $6 + 6 + 3 = 15 \Rightarrow 32\text{kB}$
- For performance: $\text{bits}(\text{CI}) + \text{bits}(\text{CO}) \leq \text{bits}(\text{VPO})$



- **Unless you have huge pages**, VPO will not be too big:
harshly limits CI+CO

Virtual Memory

- “Virtually/Physically **Indexed**” - “Virtually/Physically **Tagged**” depends on how we **access the cache**
- Virtually Indexed – Virtually Tagged
- Virtually Indexed – Physically Tagged
- **Physically Indexed – Virtually Tagged**
- Physically Indexed – Physically Tagged

Physically Indexed – Virtually Tagged

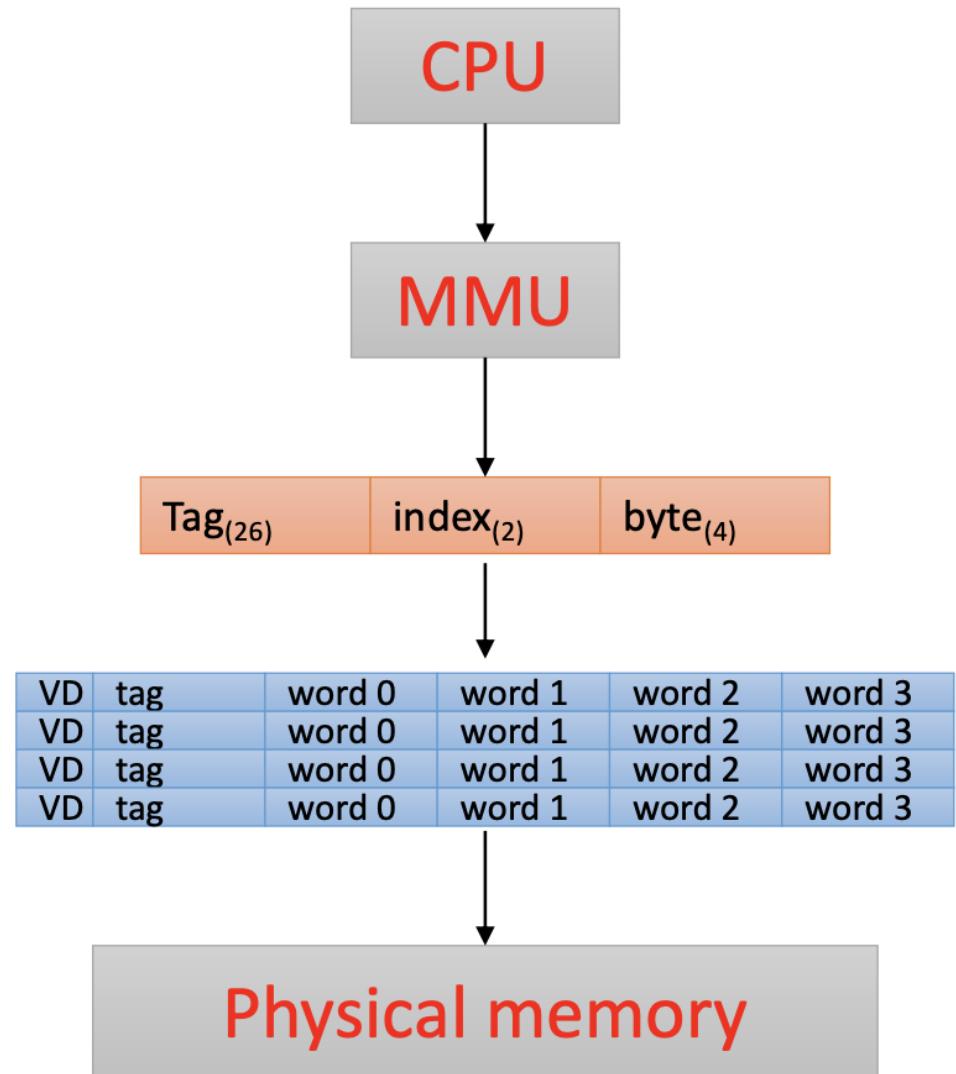
- **Makes no sense at all**
- **Physically Indexed:**
indexing is slow as we
have to wait for address
translation
- **Virtually Tagged:**
introduces homonym
and synonym issues as
seen before

Virtual Memory

- “Virtually/Physically **Indexed**” - “Virtually/Physically **Tagged**” depends on how we **access the cache**
- Virtually Indexed – Virtually Tagged
- Virtually Indexed – Physically Tagged
- Physically Indexed – Virtually Tagged
- **Physically Indexed – Physically Tagged**

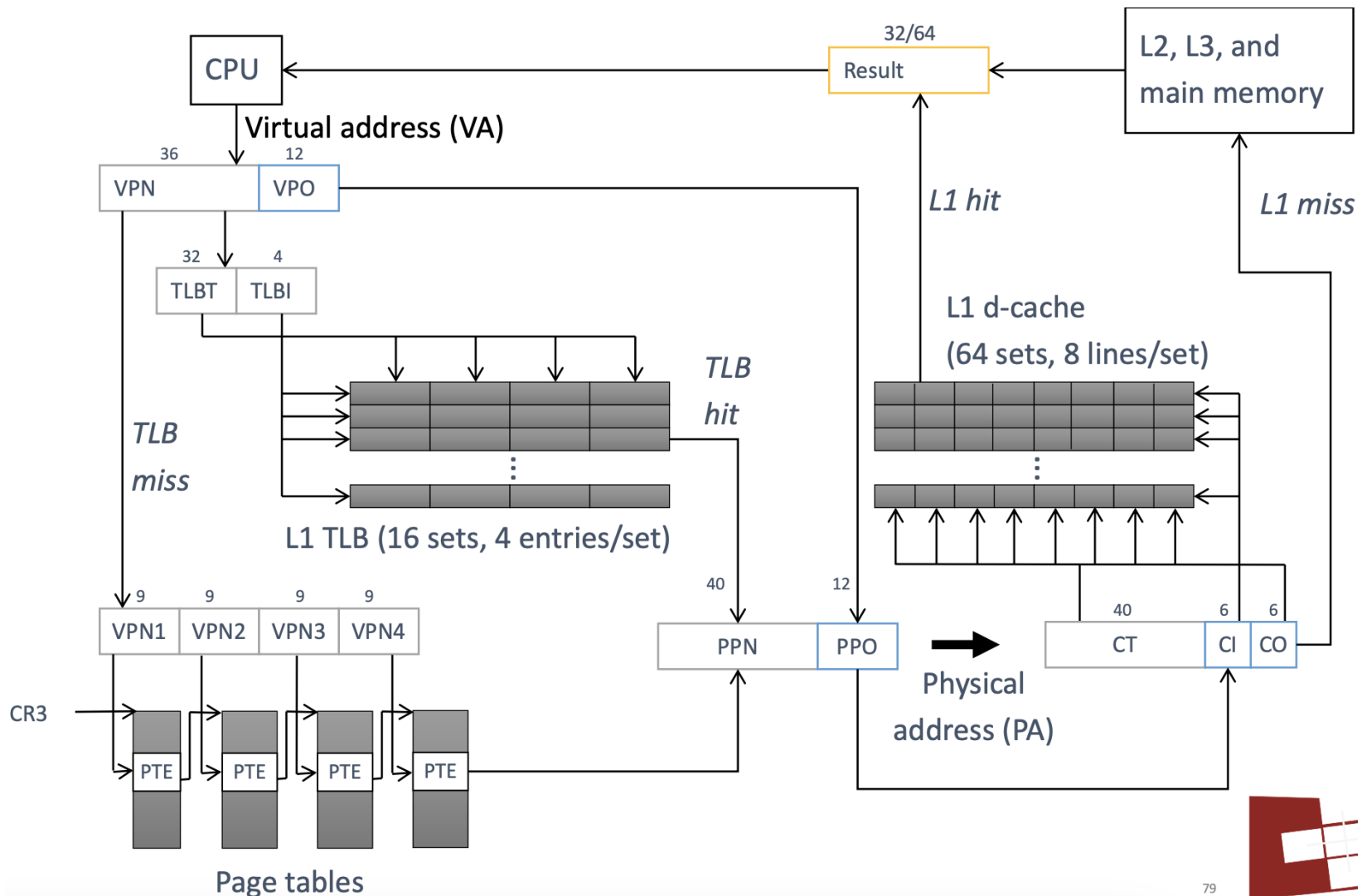
Physically Indexed – Physically Tagged

- **Slowest:** requires full address translation before lookup
- **No homonyms or synonyms**
- **Typically used for L2, L3** as we have already done the translation until then



Caches and Virtual Memory

- Is there still something unclear here?



Quiz

Quiz

Question 1

This problem requires you to analyze the cache behavior of a function that sums the elements of an array A:

```
int A[2][4];

int sum()
{
    int sum = 0;

    for (int j = 0; j < 4; j++) {
        for (int i = 0; i < 2; i++) {
            sum += A[i][j];
        }
    }
    return sum;
}
```

Assume the following:

- The memory system consists of registers, a single L1 cache, and main memory.
- The cache is cold when the function is called and the array has been initialized elsewhere.
- Variables `i`, `j` and `sum` are all stored in registers.
- The array A is aligned in memory such that the first two array elements map to the same cache block.
- `sizeof(int) == 4`.
- The cache is direct mapped, with a block size of 8 bytes.

Quiz

- a) Suppose that the cache consists of 2 sets. Fill out the table to indicate if the corresponding memory access in A will be a hit (**h**) or a miss (**m**).

A	Col 0	Col 1	Col 2	Col 3
Row 0	m			
Row 1				

Quiz

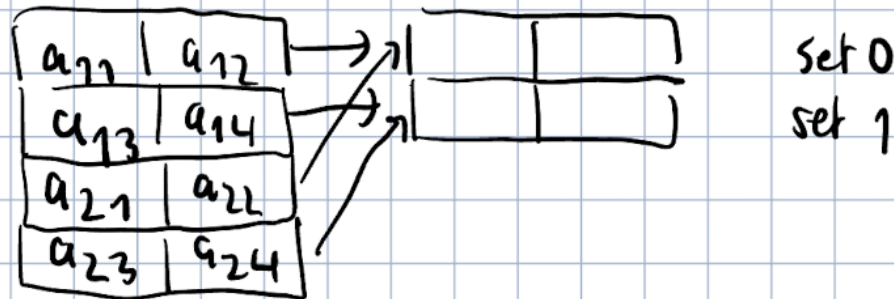
Q1a)

$A \in \mathbb{R}^{2 \times 4}$

$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

Memory

Cache



- **Cache Drawing:** One row means **one block** (I just created two cells that we can see that 2 ints can go in one cache block)!

Quiz

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = \boxed{a_{11}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} \boxed{a_{11}} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

Cache empty: it misses

Memory

Cache

<u>a_{11}</u>	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

set 0
set 1

C0

C1

C2

C3

R0
R1

m

\Rightarrow will put a_{11} in cache

Memory

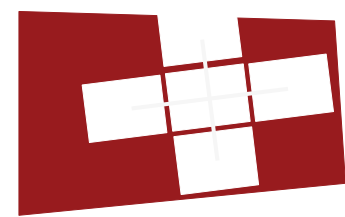
Cache

<u>a_{11}</u>	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

<u>a_{11}</u>	a_{12}

set 0
set 1

Quiz



Systems@ETH zürich

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = a_{11} + a_{21}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

Memory

Cache

a_{11}	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

a_{11}	a_{12}

set 0
set 1

Misses in cache

C0 C1 C2 C3

R0 m
R1 m

\Rightarrow will put $a_{21} | a_{22}$ in cache. **evict** old block

Memory

Cache

a_{11}	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

a_{21}	a_{22}

set 0
set 1

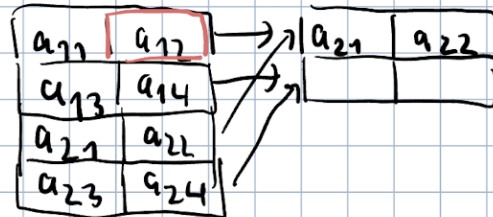
Quiz

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = a_{11} + a_{21} + \boxed{a_{12}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} a_{11} & \boxed{a_{12}} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

Memory

Cache



Set 0
Set 1

R0
R1

Misses in cache

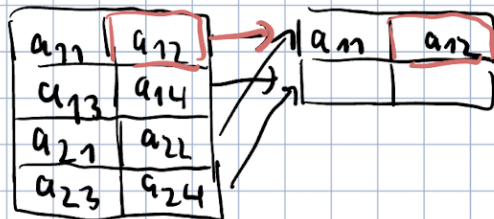
C0 C1 C2 C3

m m M

\Rightarrow will put a_{11} | a_{12} in cache. **evict** old block

Memory

Cache



Set 0
Set 1

Quiz

- You get the idea ...

A	Col 0	Col 1	Col 2	Col 3
Row 0	m	m	m	m
Row 1	m	m	m	m

- The **underlying issue**: cache too small, forces overlapping cache accessing
- What would be a possible solution to get higher cache hit rate **without increasing cache size**?

Quiz

b) What is the pattern of hits and misses if the cache consists of 4 sets instead of 2 sets?

A	Col 0	Col 1	Col 2	Col 3
Row 0	m			
Row 1				

Quiz

Q1b)

$A \in \mathbb{R}^{2 \times 4}$

$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

Memory

Cache

a_{11}	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

set 0

set 1

set 2

set 3

Quiz

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = \boxed{a_{11}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} \boxed{a_{11}} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

Cache empty: it misses

Memory

Cache

$\boxed{a_{11}}$	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

set 0
set 1
set 2
set 3

R0
R1

C0

C1

C2

C3

\boxed{m}

\Rightarrow will put $\boxed{a_{11} | a_{12}}$ in cache

Memory

Cache

$\boxed{a_{11}}$	a_{12}
a_{13}	a_{14}
a_{21}	a_{22}
a_{23}	a_{24}

$\boxed{a_{11}}$	a_{12}

set 0
set 1
set 2
set 3

Quiz

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = a_{11} + \boxed{a_{21}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \boxed{a_{21}} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

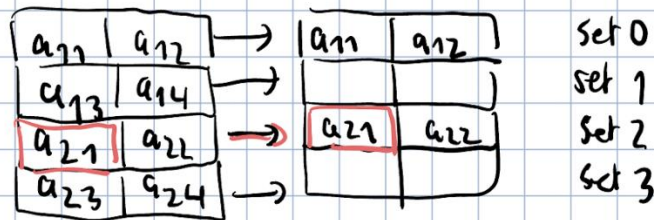
Memory Cache



Misses in cache

	C0	C1	C2	C3
R0	m			
R1	\boxed{m}			

⇒ will put $\boxed{a_{21} | a_{22}}$ in cache.



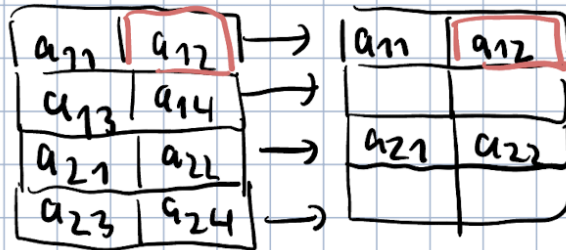
Quiz

Access pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = a_{11} + a_{21} + \boxed{a_{12}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} a_{11} & \boxed{a_{12}} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$

Memory

Cache



set 0
set 1
set 2
set 3

R0
R1

Hits in cache

C0

C1

C2

C3

m
n

\boxed{h}

Quiz

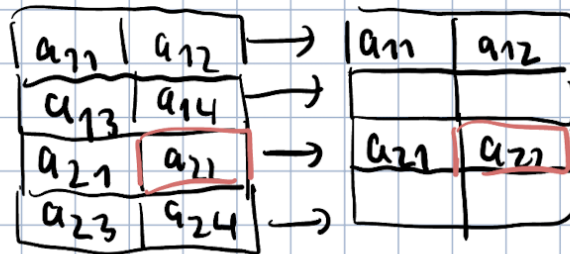
Accum pattern: $\sum_{j=1}^4 \sum_{i=1}^2 a_{ij} = a_{11} + a_{21} + a_{12} + \boxed{a_{22}}$

$A \in \mathbb{R}^{2 \times 4} = \begin{pmatrix} a_{11} & \boxed{a_{12}} & a_{13} & a_{14} \\ a_{21} & \boxed{a_{22}} & a_{23} & a_{24} \end{pmatrix}$

Hits in cache

Memory

Cache



set 0
set 1
set 2
set 3

RO	m	h
R1	m	\boxed{h}

Physically Indexed – Virtually Tagged

- Next accesses analogous

A	Col 0	Col 1	Col 2	Col 3
Row 0	m	h	m	h
Row 1	m	h	m	h

- Thus we have seen we can increase hit rate by increase **cache size** (next to doing row accesses)

Quiz

Question 2

This problem tests your understanding of the cache organization and performance. Assume the following:

- `sizeof(int) == 4`
- Array `x` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `x`. All variables are stored in registers.

Consider the following C code:

```
int x[128];
int j;
int sum = 0;

for (int i = 0; i < 64; i++) {
    j = i + 64;
    sum += x[i] * x[j];
}
```

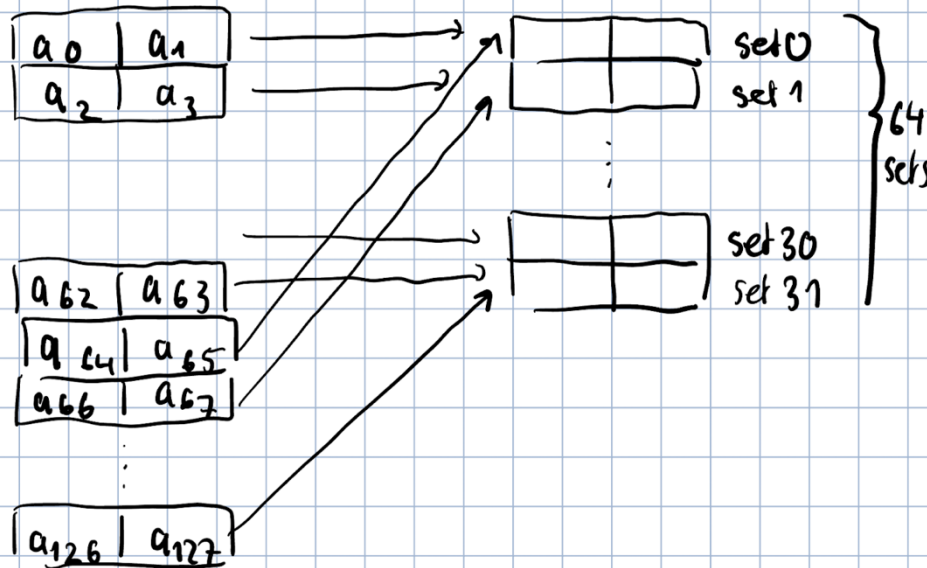
Quiz

Case 1

- a) Assume your cache is a 256-byte directed-mapped data cache with 8-byte cache blocks. What is the cache **miss rate**?

Quiz

Q2a)

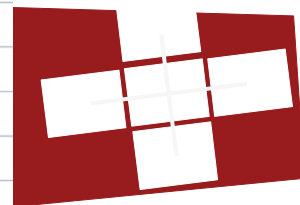


$$C = 256 \text{ Byte}$$

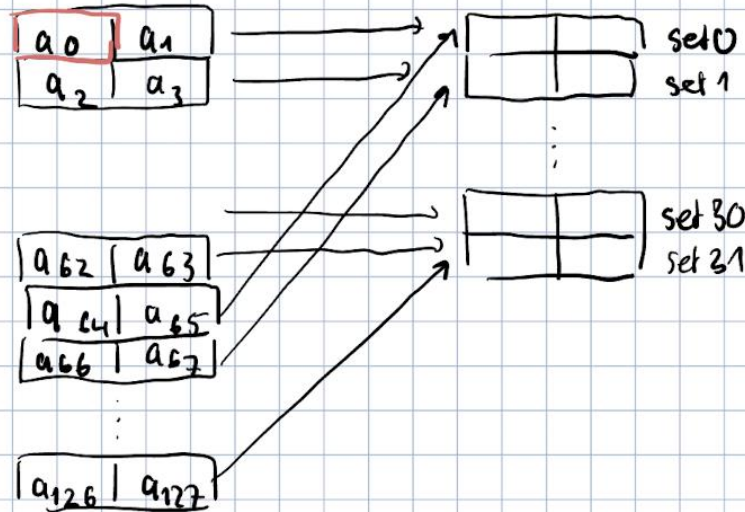
$$B = \frac{256 \text{ Byte}}{8 \text{ Byte}} = 32 \text{ blocks} \Rightarrow 32 \text{ sets}$$

Access pattern: $\sum_{i=0}^{63} x(i) \cdot x(i+64) = x(0) + x(64)$

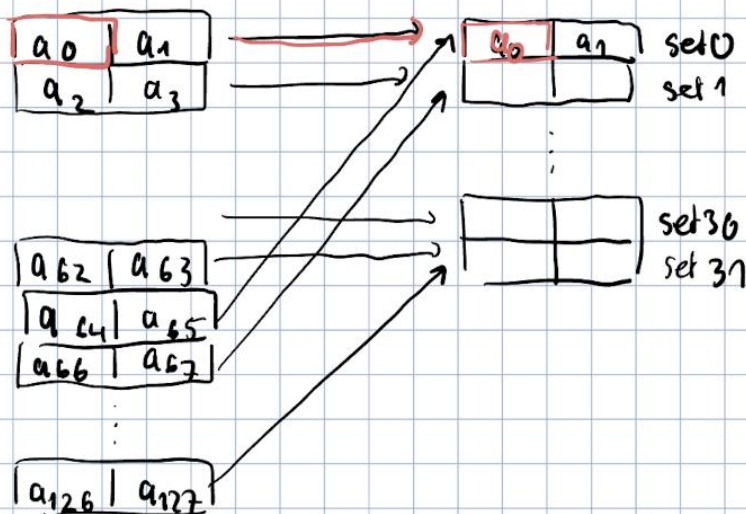
cache miss



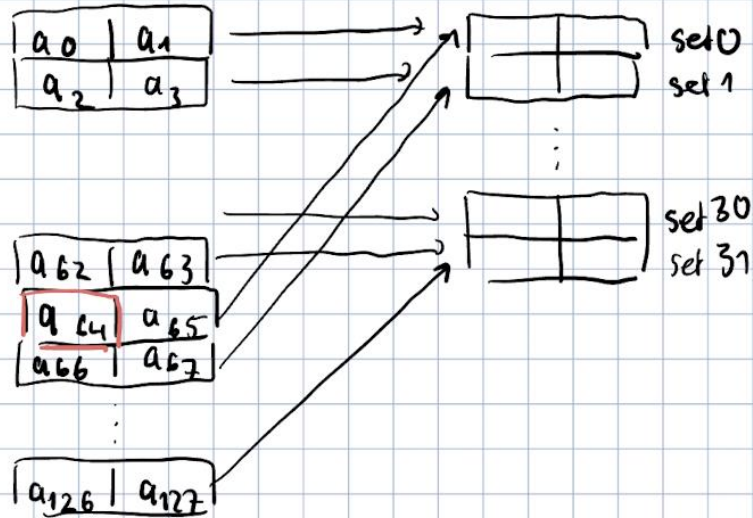
Systems@ETH Zürich



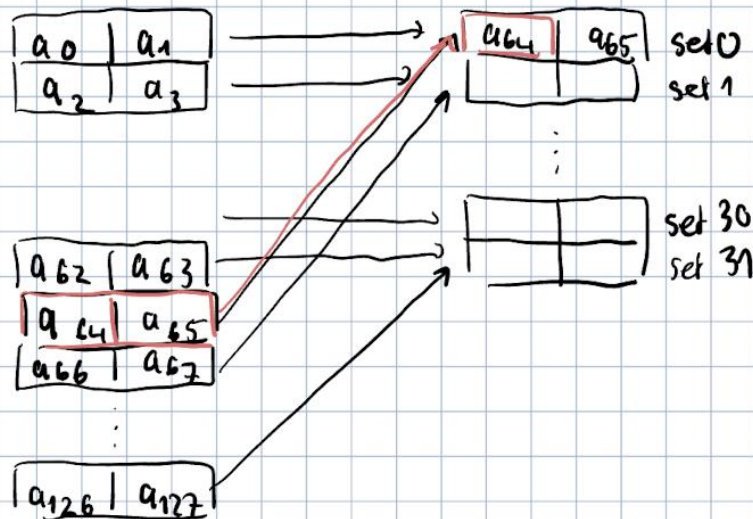
→ Put block $\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix}$ in cache



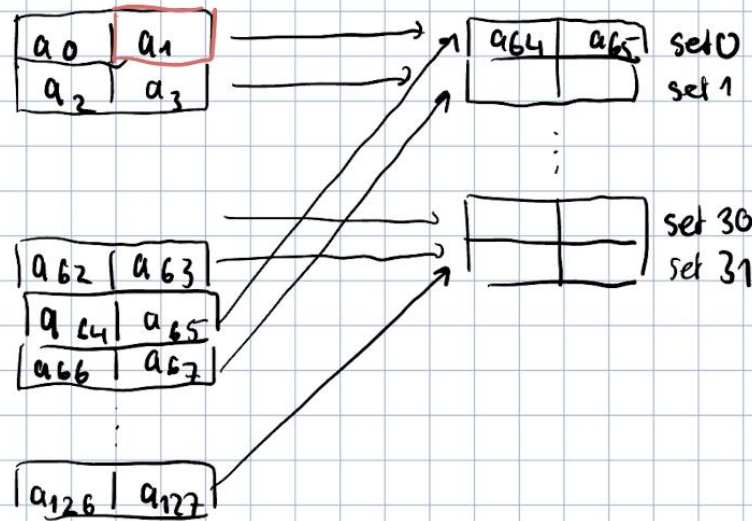
Accum pattern: $\sum_{i=0}^{63} x[i] \cdot x[i+64] = x[0] + x[64]$



⇒ Put block $a_{64} | a_{65}$ in cache, **evict** $a_0 | a_1$



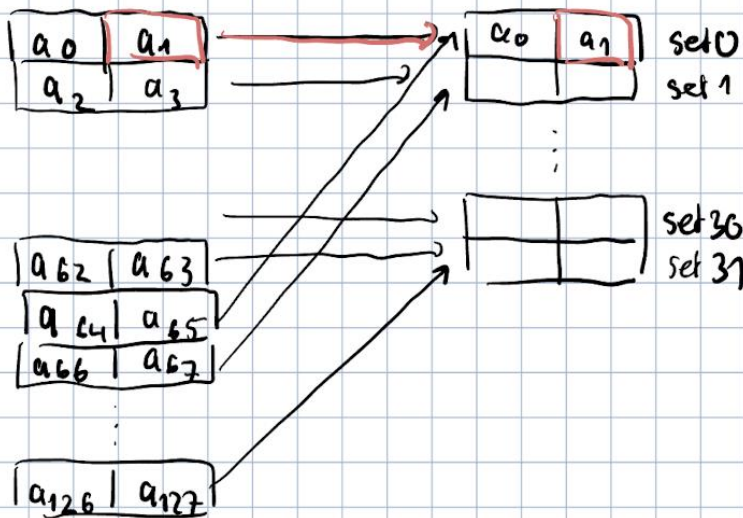
Access pattern: $\sum_{i=0}^{63} x(i) \cdot x(i+64) = x(0) + x(64) + \boxed{x(1)} + x(65)$



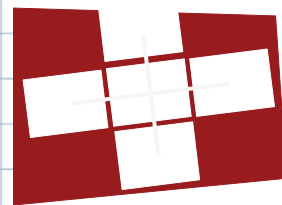
Cache miss

systems@ETH zürich

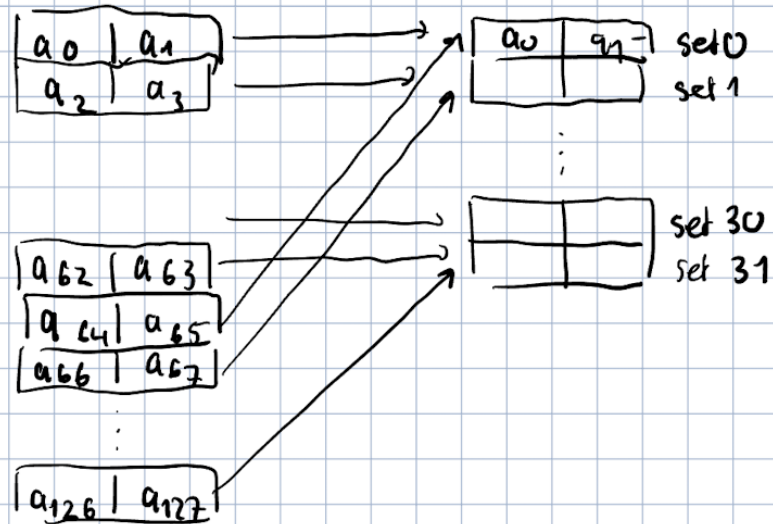
⇒ Put block a_0, a_1 in cache, evict a_{64}, a_{65}



Access pattern: $\sum_{i=0}^{63} x[i] \cdot x[i+64] = x[0] + x[64] + x[1] + x[65]$

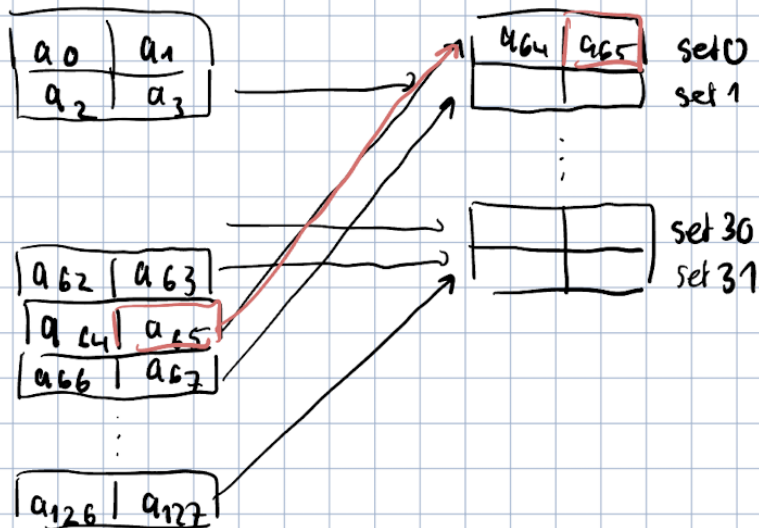


tems@ETH zürich



Cache miss

⇒ Put block a64/a65 in cache, evict a0/a1



Quiz

- **Pattern continues**

- a) Assume your cache is a 256-byte directed-mapped data cache with 8-byte cache blocks. What is the cache **miss rate**?

Solution: 100%

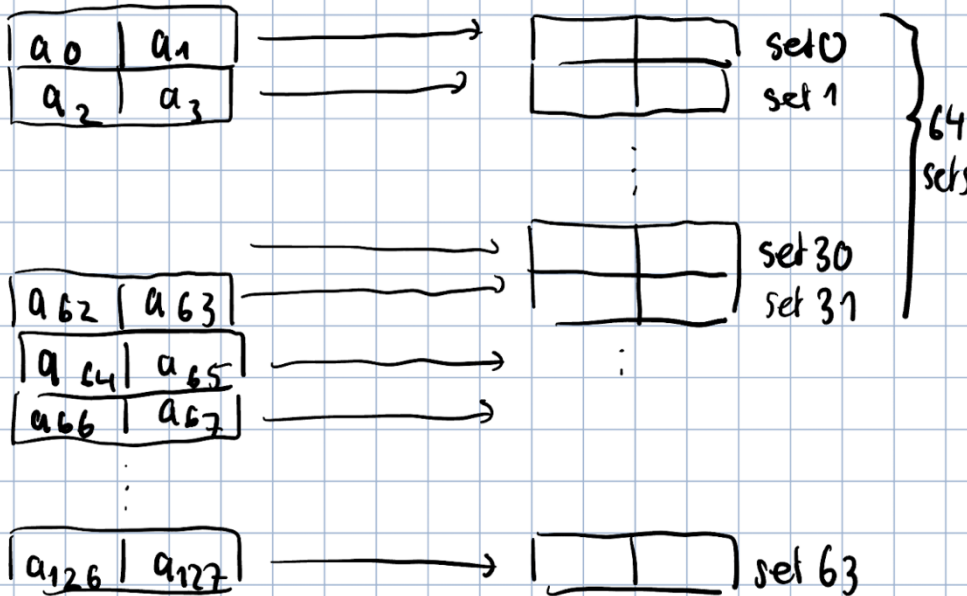
Quiz

b) If the cache were twice as big, what would be the miss rate?

Solution: 50%

Quiz

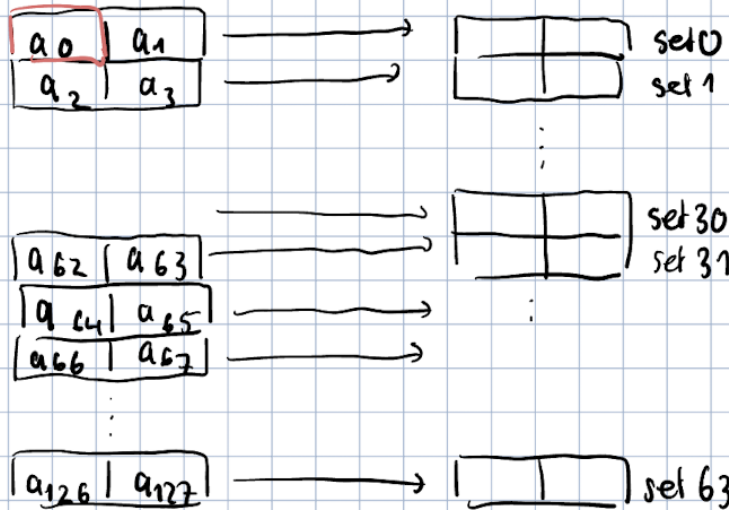
Q261



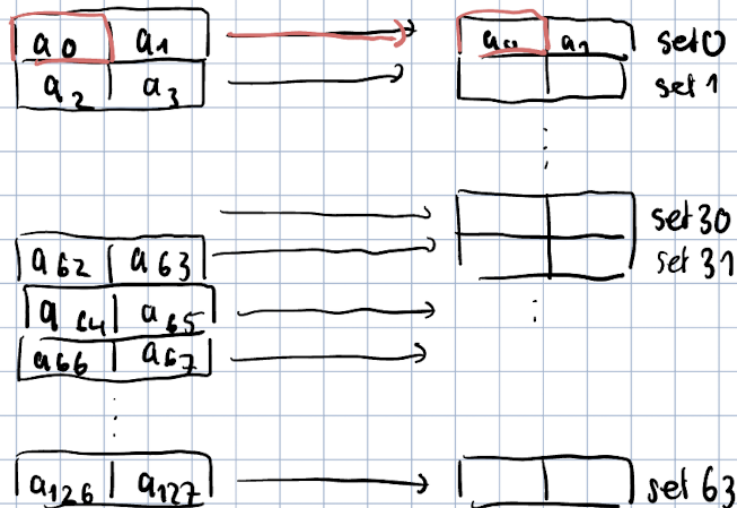
$$C = 512 \text{ Byte}$$
$$B = \frac{512 \text{ Byte}}{8 \text{ Byte}} = 64 \text{ Blocks}$$
$$\Rightarrow 64 \text{ sets}$$

Access pattern: $\sum_{i=0}^{63} x(i) \cdot x(i+64) = x(0) + x(64)$

cache miss

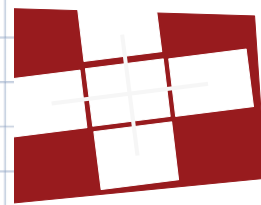


⇒ Put block $a_0 | a_1$ in cache

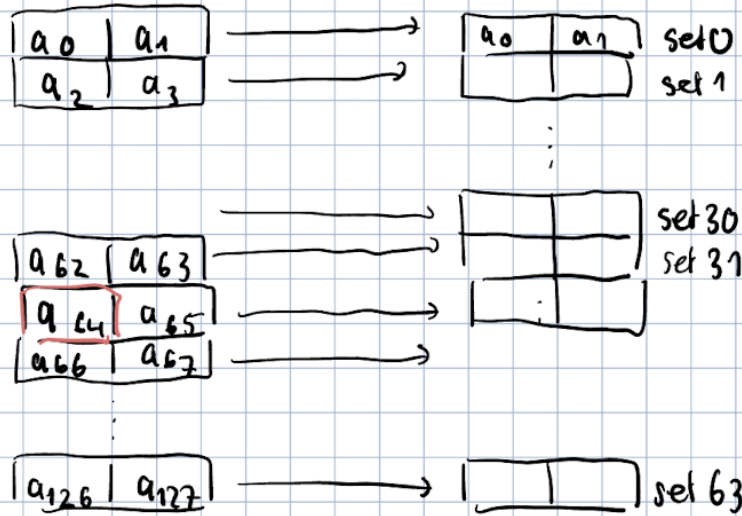


Access pattern: $\sum_{i=0}^{63} x(i) \cdot x(i+64) = x(0) + x(64)$

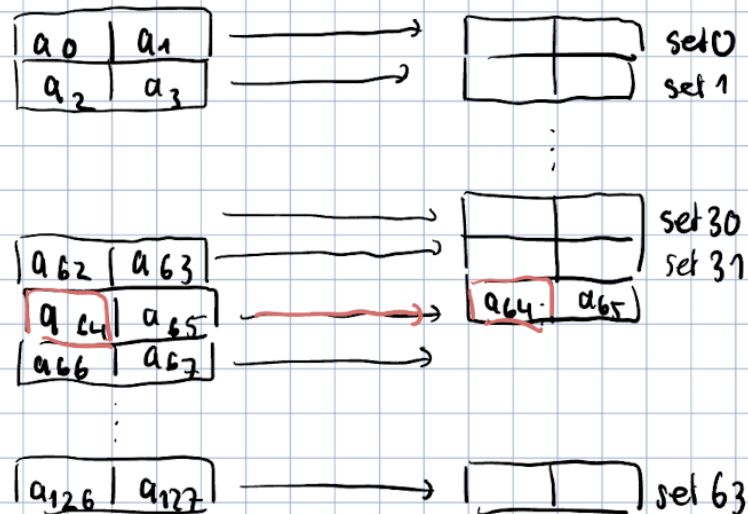
cache miss



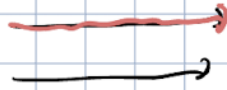
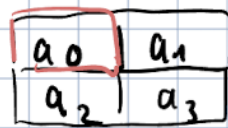
ms@ETH zürich



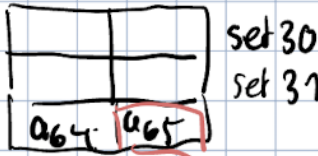
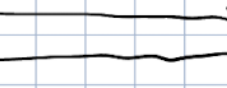
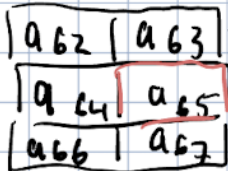
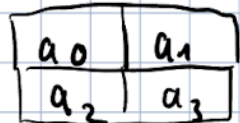
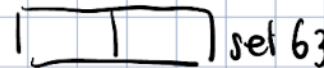
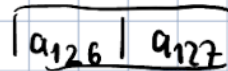
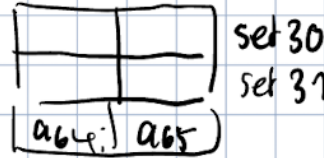
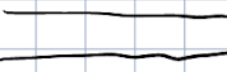
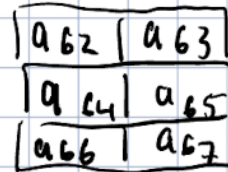
⇒ Put block a_{64}, a_{65} in cache



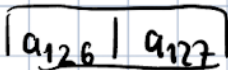
Access pattern: $\sum_{i=0}^{63} x[i] \cdot x[i+64] = x[0] + x[64] + x[1] + x[65]$



hit



hit



Quiz

- **Pattern continues:** Compulsory Miss, Compulsory Miss, Hit, Hit

Case 1

- a) Assume your cache is a 256-byte directed-mapped data cache with 8-byte cache blocks. What is the cache **miss rate**?

Solution: 100%

- b) If the cache were twice as big, what would be the miss rate?

Solution: 50%

Quiz - Remark

- **In an exam:** you don't have as much time, you sketch or just calculate with the numbers!!

Case 1

- a) Assume your cache is a 256-byte directed-mapped data cache with 8-byte cache blocks. What is the cache **miss rate**?
- b) If the cache were twice as big, what would be the miss rate?

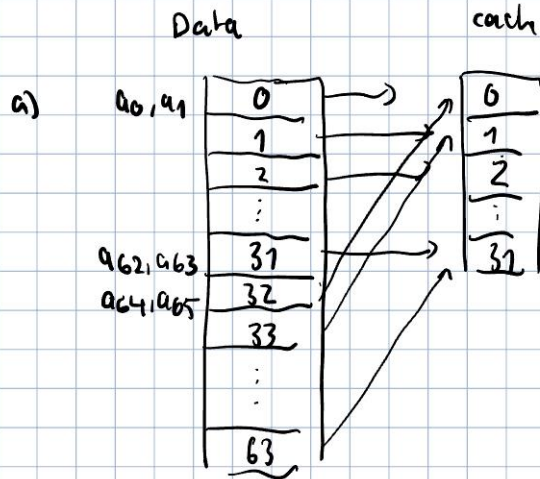
Quiz - Remark

Calculation

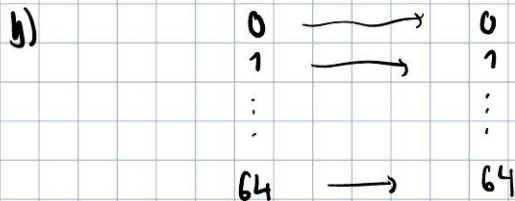
Data: $128 \text{ entries} \cdot 4 \text{ bytes} = 512 \text{ bytes}$

$\frac{512 \text{ bytes}}{8 \text{ byte}} = 64 \text{ blocks}$

Cache: $\frac{256 \text{ bytes}}{8 \text{ byte}} = 32 \text{ blocks}$
 $\Rightarrow 32 \text{ sets}$



100 % miss rate



miss rate: 50 %

(one cold miss per 1st access)

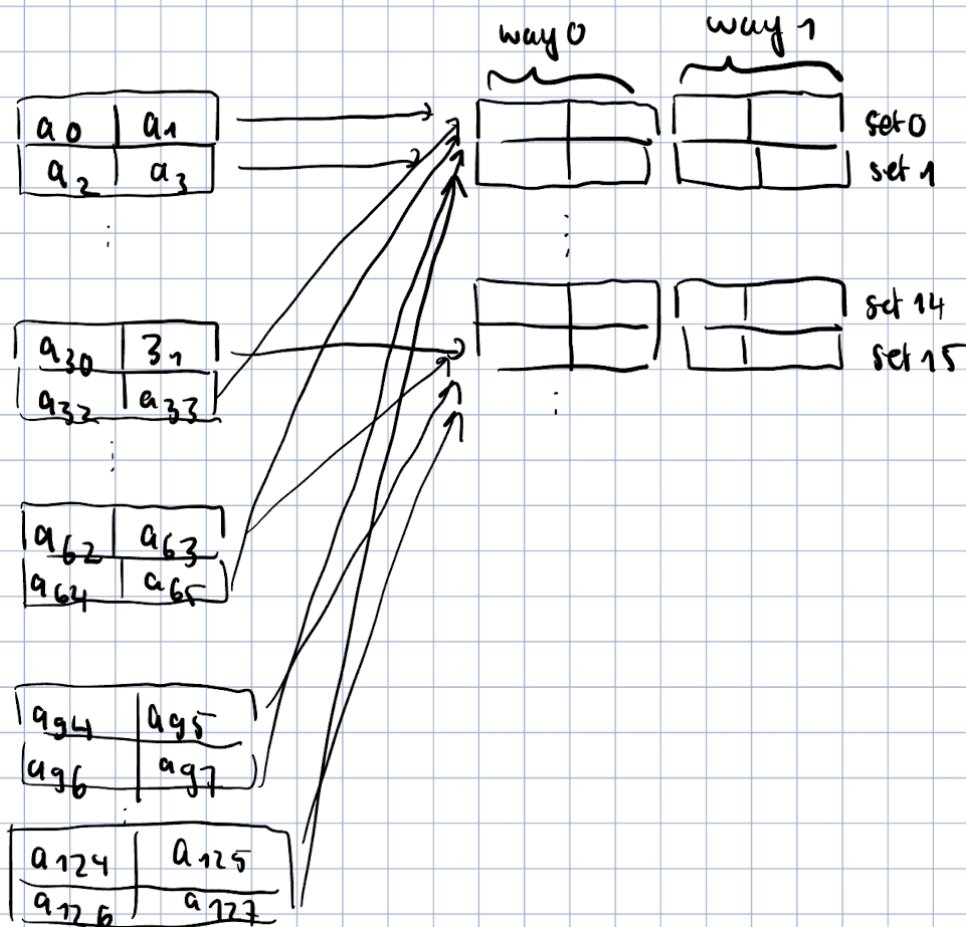
Quiz

Case 2

- a) Assume your cache is 256-byte 2-way set associative using an LRU replacement policy with 8-byte cache blocks. What is the cache miss rate?
- b) Will larger **cache size** help to reduce the miss rate?
- c) Will larger **cache line** help to reduce the miss rate?

Quiz

(2.2. a)



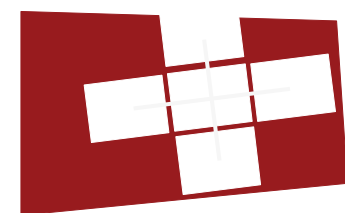
$$\begin{aligned}
 C &= 256 \text{ Byte} \\
 B &= \frac{256 \text{ Byte}}{8 \text{ Byte}} = 32 \text{ Blocks} \\
 &\Rightarrow \frac{32}{2} = 16 \text{ Sets}
 \end{aligned}$$

64 sets

Quiz

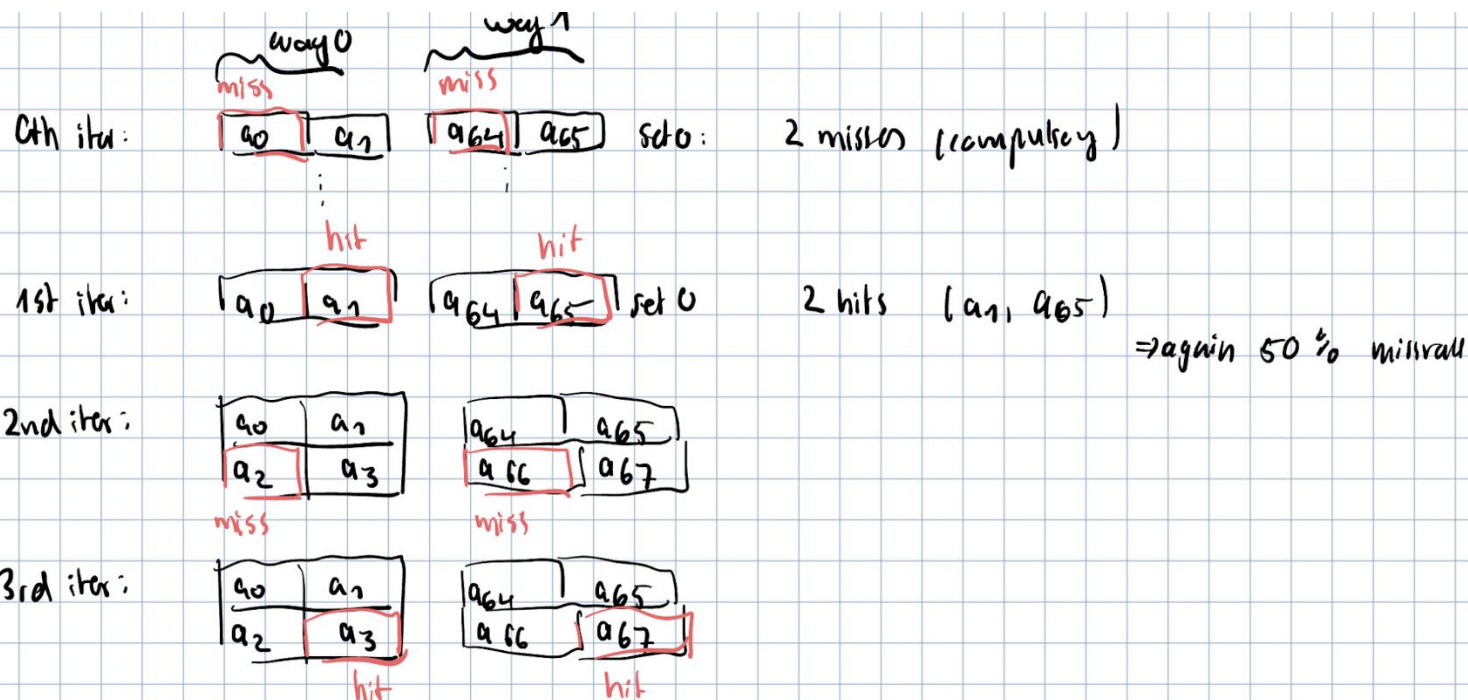
- a) Assume your cache is 256-byte 2-way set associative using an LRU replacement policy with 8-byte cache blocks. What is the cache miss rate?
- b) Will larger **cache size** help to reduce the miss rate?
- c) Will larger **cache line** help to reduce the miss rate?

Quiz



Systems@ETH zürich

- a) Assume your cache is 256-byte 2-way set associative using an LRU replacement policy with 8-byte cache blocks. What is the cache miss rate?
- b) Will larger **cache size** help to reduce the miss rate?



b) Larger cache size: would not help

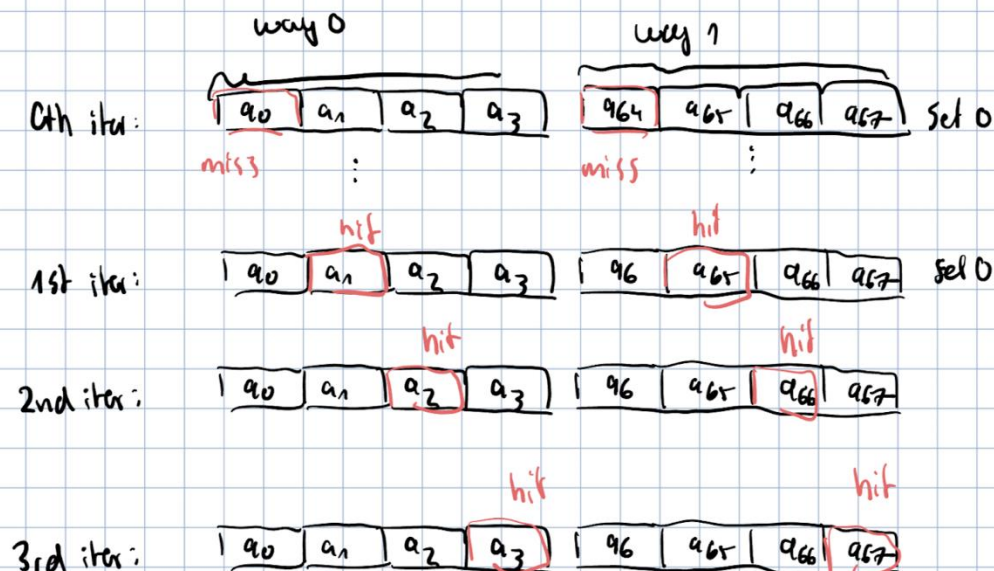
Quiz

c) Will larger **cache line** help to reduce the miss rate?

Quiz

c) Will larger **cache line** help to reduce the miss rate?

c) larger cache line/block size: yes, assume 16 byte block size



Miss rate: $\frac{1}{\text{\#elements per block}}$

next $\Rightarrow \frac{1}{4}$ miss rate

before $\Rightarrow \frac{1}{2}$ miss rate

Quiz

Case 2

- a) Assume your cache is 256-byte 2-way set associative using an LRU replacement policy with 8-byte cache blocks. What is the cache miss rate?

Solution: 50%

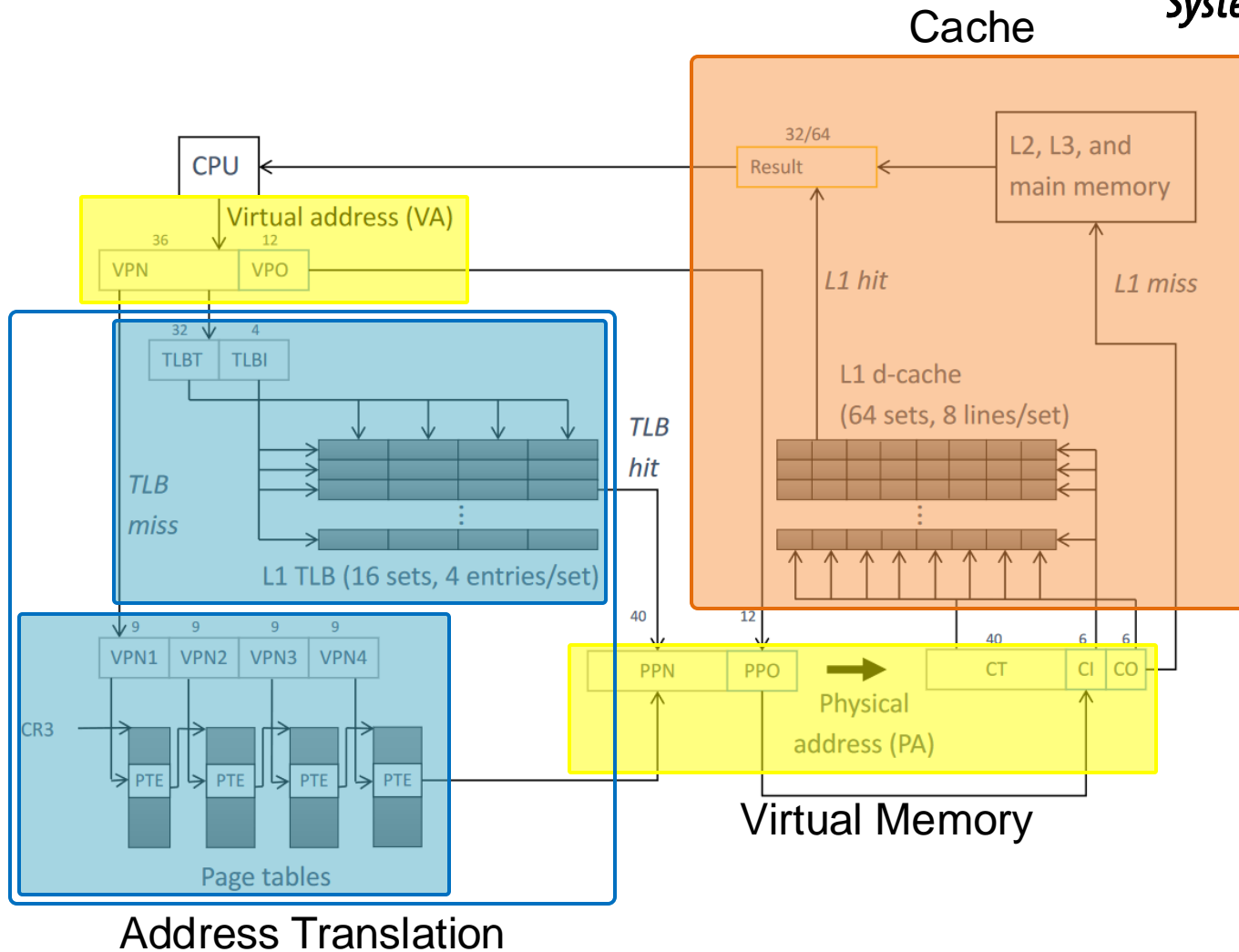
- b) Will larger **cache size** help to reduce the miss rate?

Solution: No

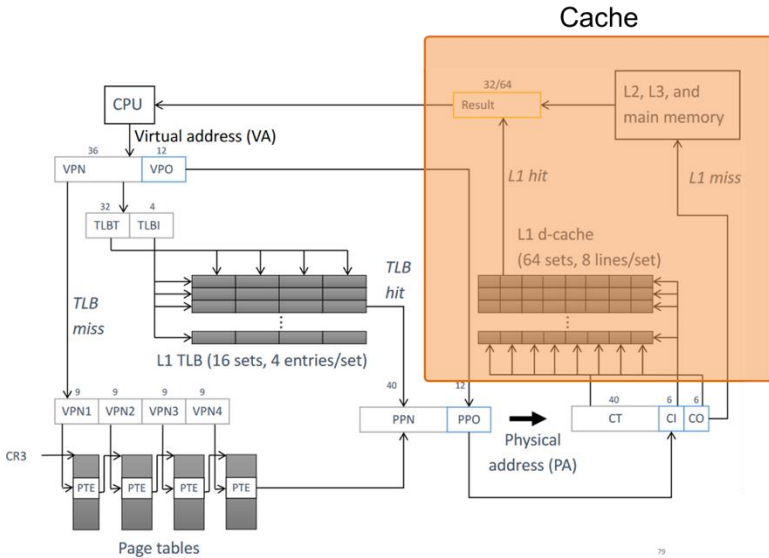
- c) Will larger **cache line** help to reduce the miss rate?

Solution: Yes

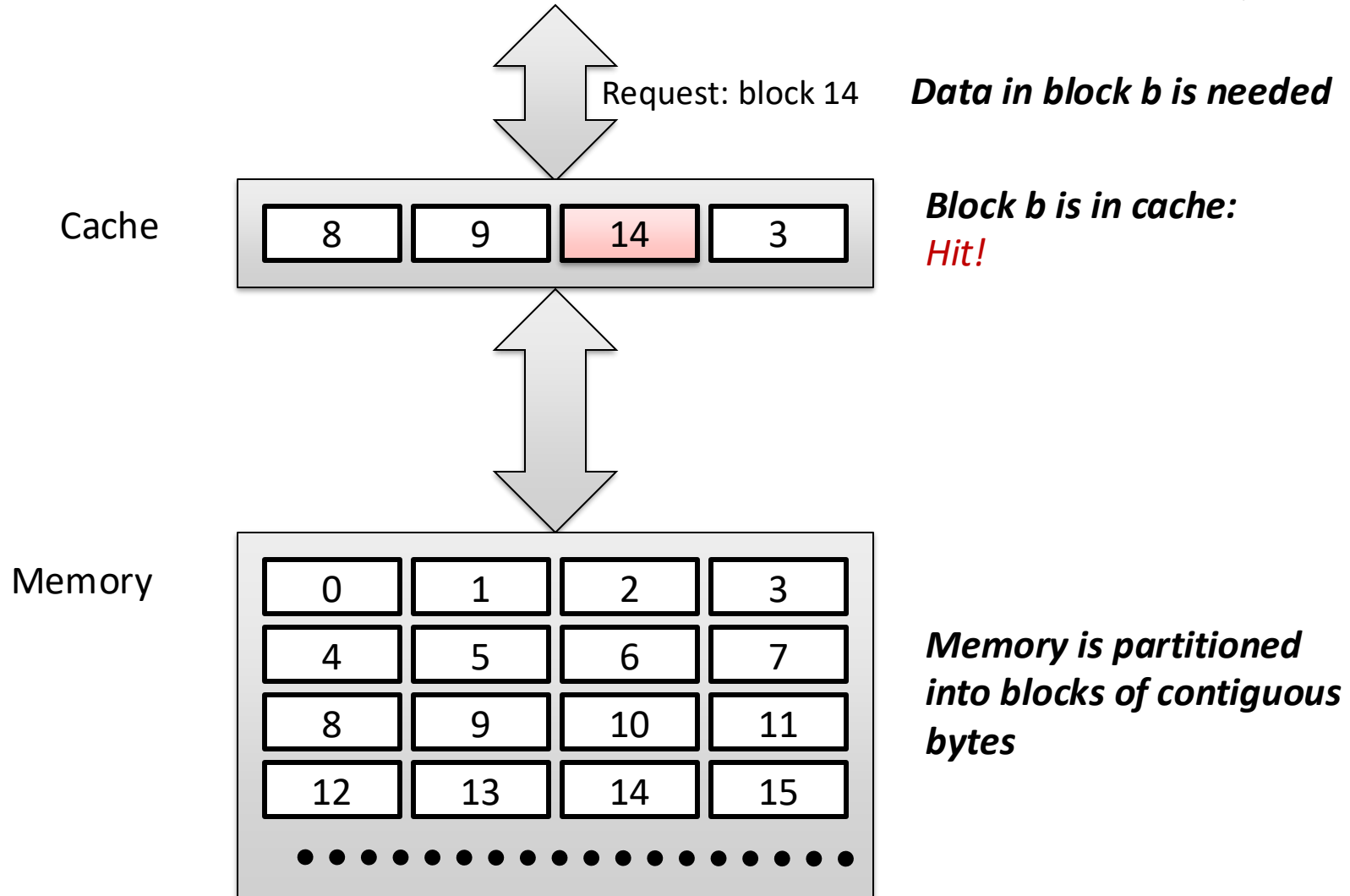
Core i7 memory system



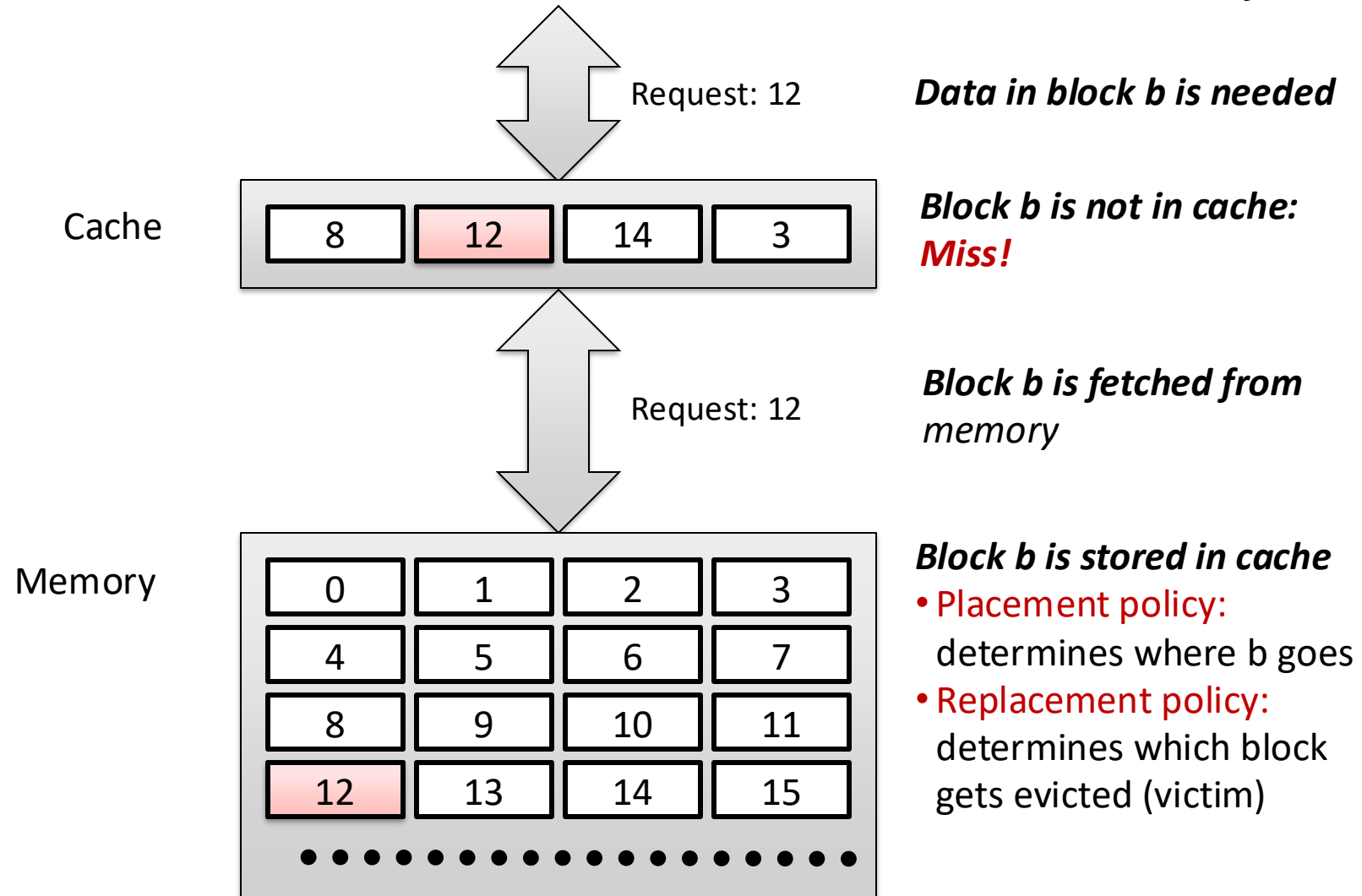
Caches



General cache concepts: Hit



General cache concepts: Miss



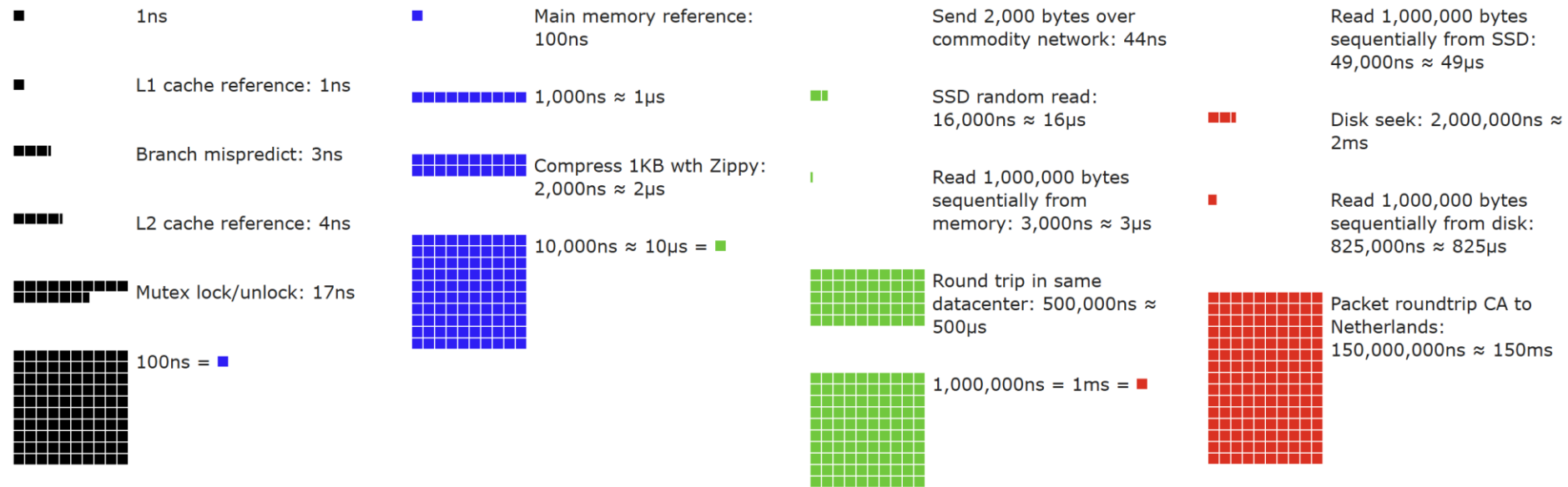
Let's think about those numbers

- Huge difference between a hit and a miss
 - Could be 100x, if just L1 and main memory
- Would you believe 99% hits is twice as good as 97%?
 - Consider:
cache hit time of 1 cycle
miss penalty of 100 cycles
 - Average access time:
97% hits: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = 4 \text{ cycles}$
99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = 2 \text{ cycles}$
- This is why “miss rate” is used instead of “hit rate”

Aside: Latency numbers

Latency Numbers Every Programmer Should Know

2020

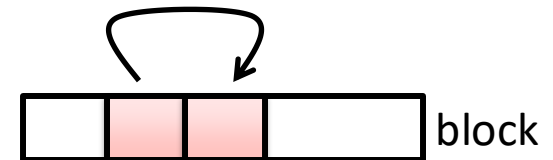


Types of cache miss

- Cold (compulsory) miss
 - Occurs on first access to a block
- Conflict miss
 - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
 - e.g., block i must be placed in slot $(i \bmod 4)$
 - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
 - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
- Capacity miss
 - Occurs when the set of active cache blocks (working set) is larger than the cache
- Coherency miss
 - Multiprocessor systems: see later in the course

Why caches work

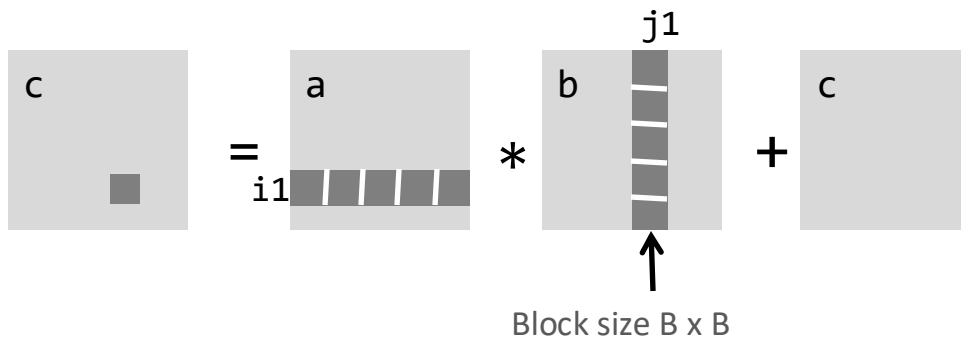
- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
- **Temporal locality:**
 - Recently referenced items are likely to be referenced again in the near future
- **Spatial locality:**
 - Items with nearby addresses tend to be referenced close together in time




Example: Blocked matrix multiplication

```
c = (double *) calloc(sizeof(double), n*n);

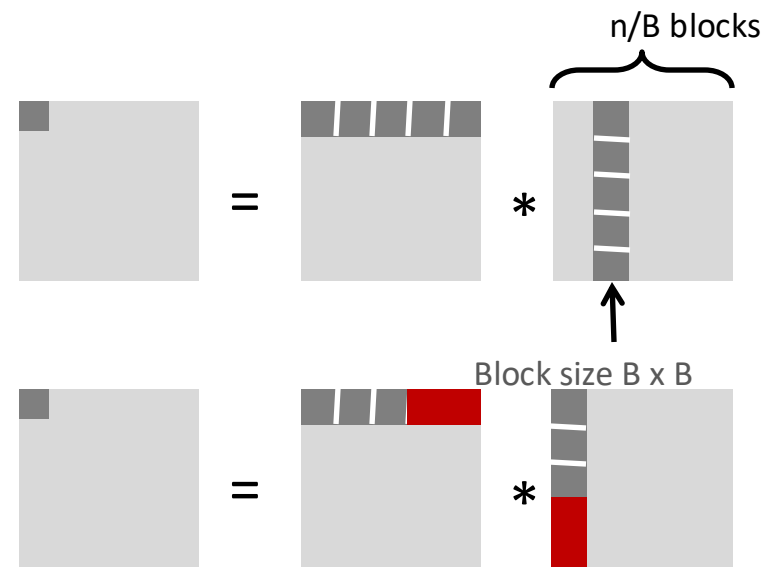
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```




Cache miss analysis

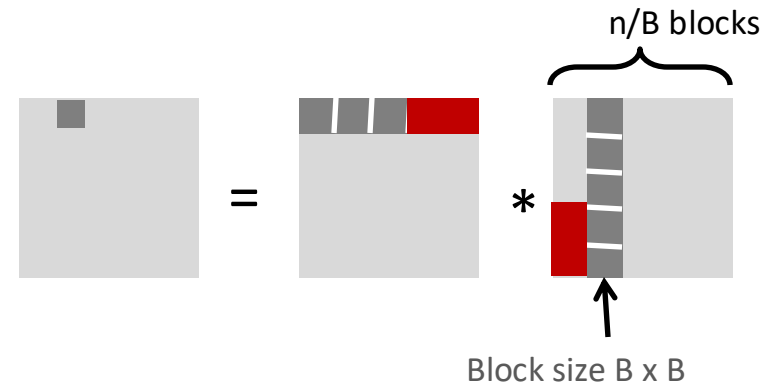
- Assume:
 - Cache block = 8 doubles = 64 Bytes
 - Cache size $C \ll n$ (much smaller than n)
 - Three blocks  fit into cache: $3B^2 < C$
 - **B is a multiple of cache block**

- First (block) iteration:
 - $B^2/8$ misses for each block
 - B rows, $B/8$ misses per row
 - $2n/B * B^2/8 = nB/4$
 - n/B blocks in a and b resp.
 - omitting matrix c, as its misses are unaffected by blocking if filled in row-major order




Cache miss analysis

- Assume:
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (much smaller than n)
 - Three blocks  fit into cache: $3B^2 < C$
 - **B is a multiple of cache block**
- Second (block) iteration:
 - Same as first iteration
 - $2n/B * B^2/8 = nB/4$
- Total misses:
 - $nB/4 * (n/B)^2 = n^3/(4B)$
 - $(n/B)^2$ blocks in c to compute






Cache miss analysis – different assumption

- Assume:
 - Cache block = 8 doubles
 - Cache size $C \ll n$ (much smaller than n)
 - Three blocks  fit into cache: $3B^2 < C$

Now if $B \leq$ cache block...

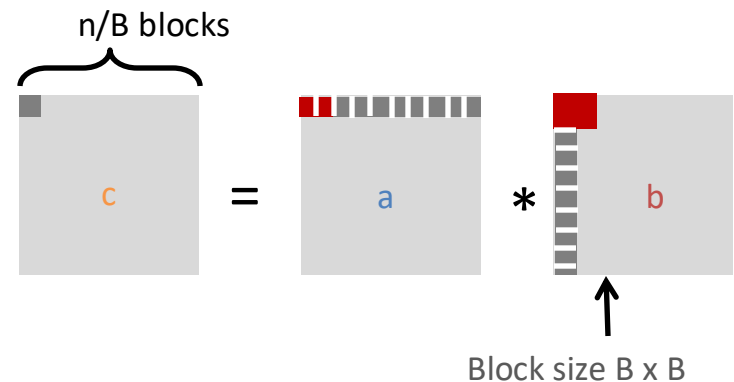
- Total misses:

$$(\text{Bn}/8 + \text{n}) * (\text{n}/\text{B})^2$$

 Misses in matrix a per iteration
  Misses in matrix b per iteration
  Number of iterations

$$= (1/(8B) + 1/B^2) * n^3$$

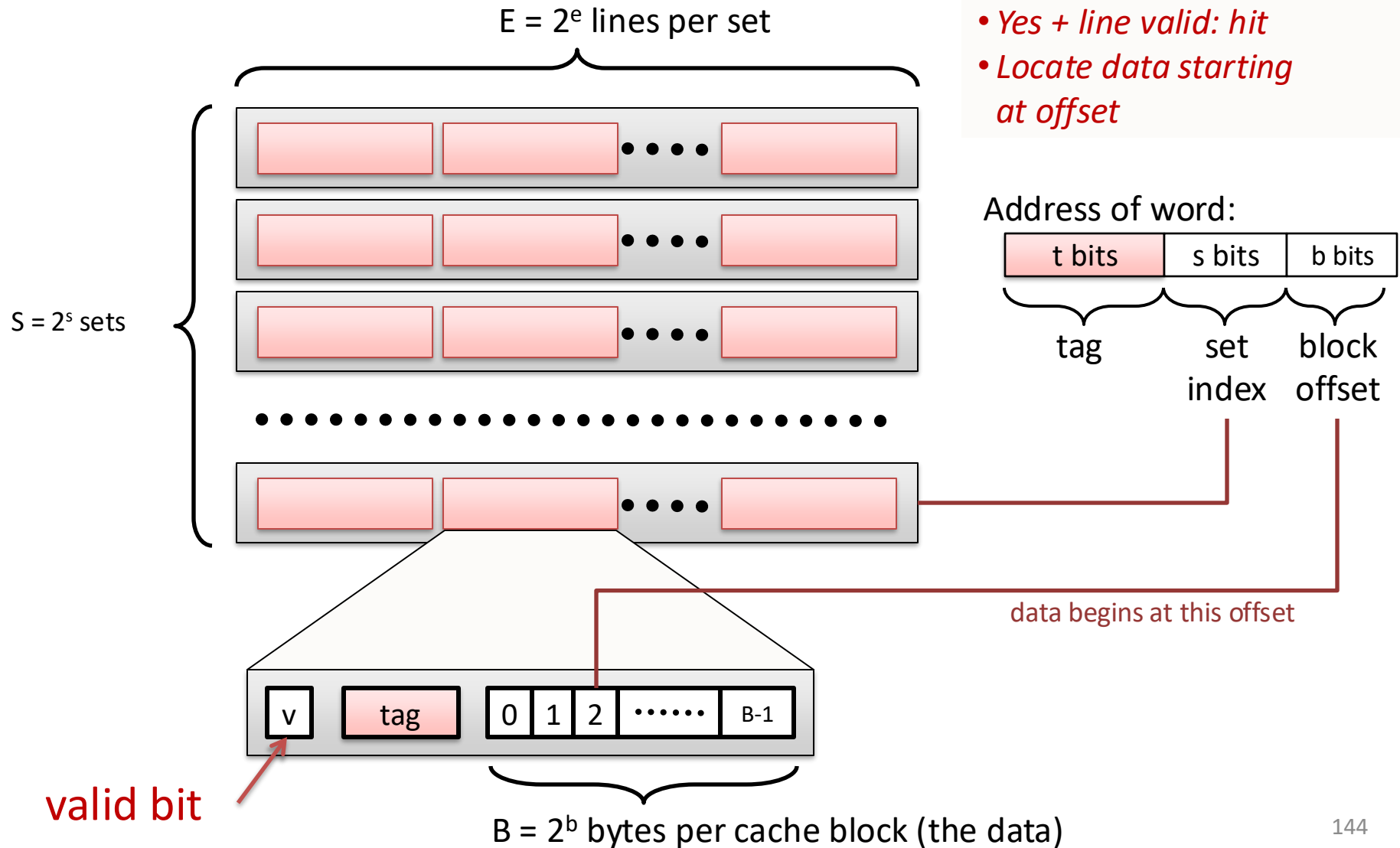
(omitting matrix c)



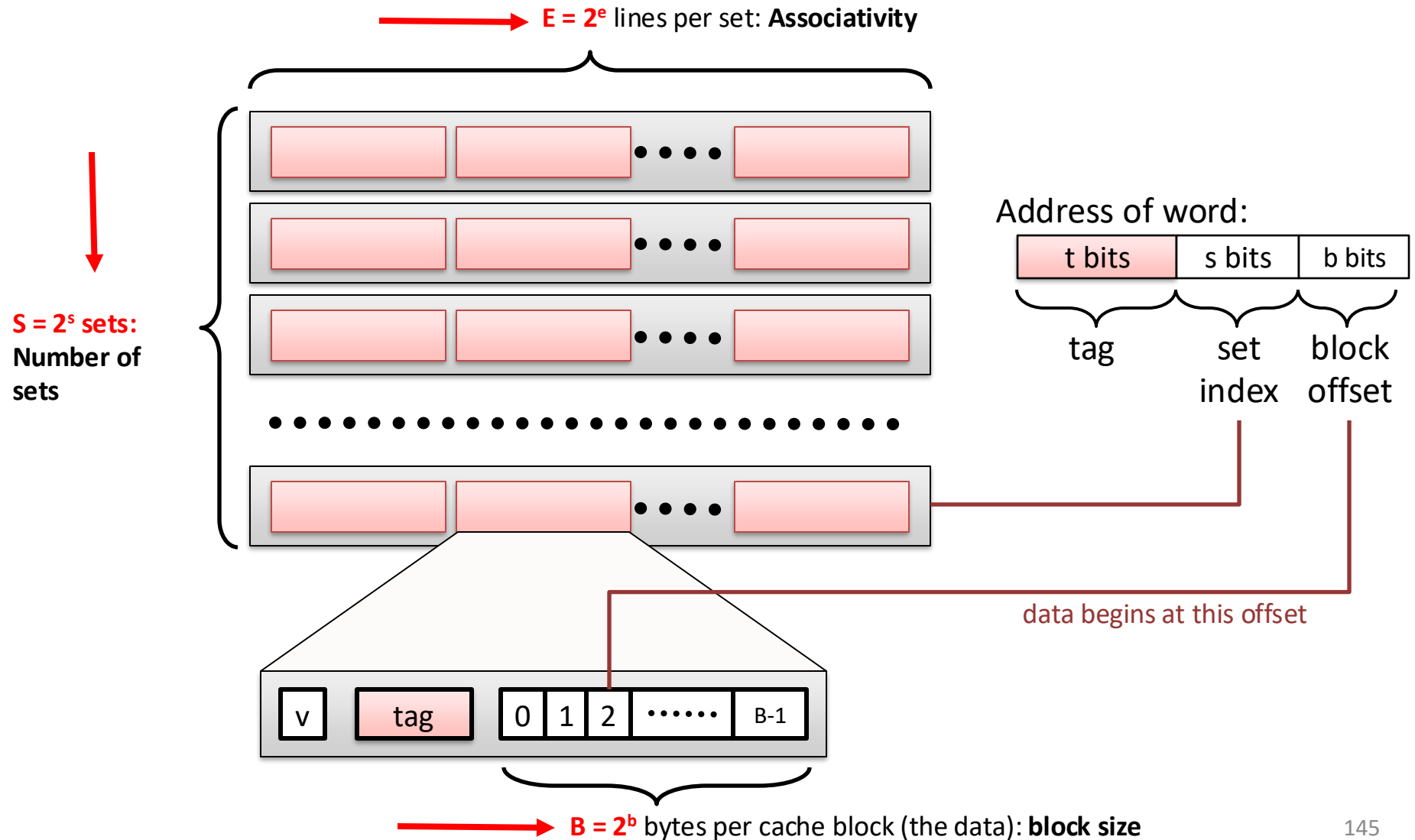
Note: if $B = 1$, same as no blocking analysis in lecture slides: $(9/8) * n^3$ misses
 if $B = 8 =$ cache block size, we have $n^3/32$ misses $= n^3/(4B)$

Cache read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

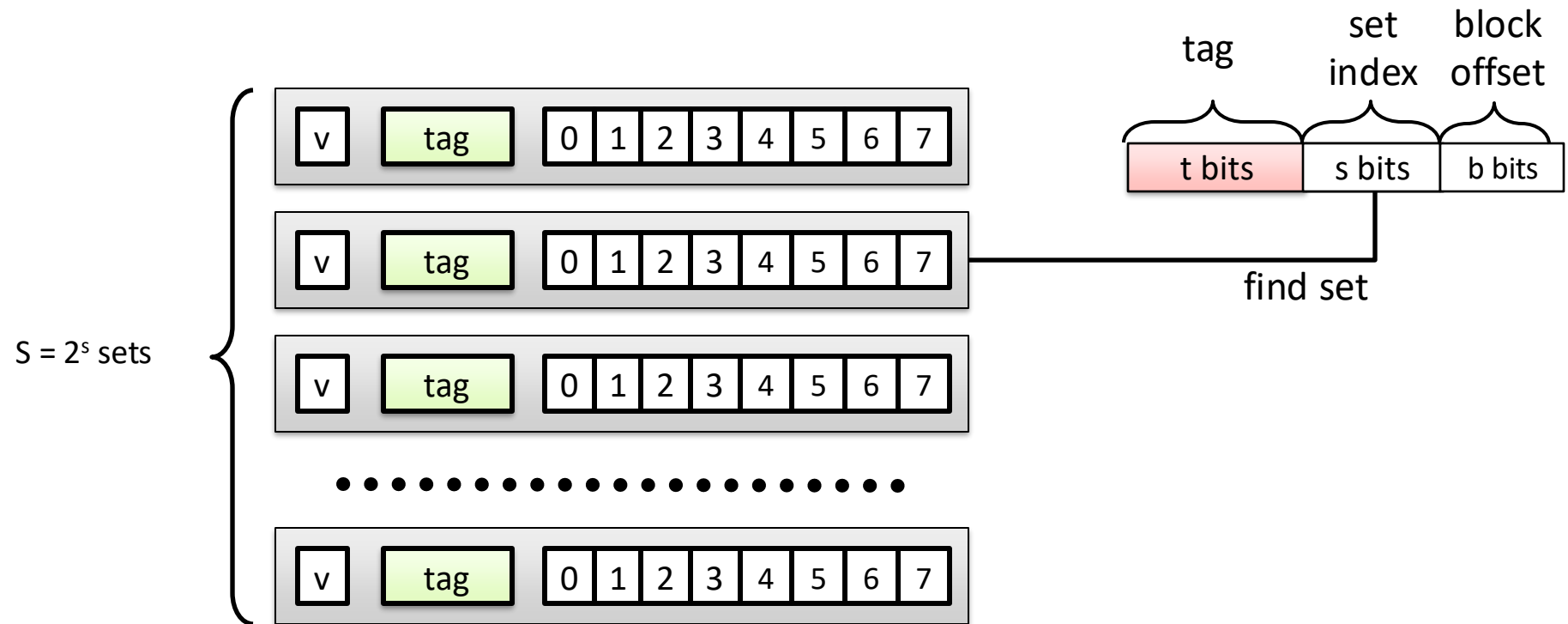


Cache parameters



Direct mapped cache ($E = 1$)

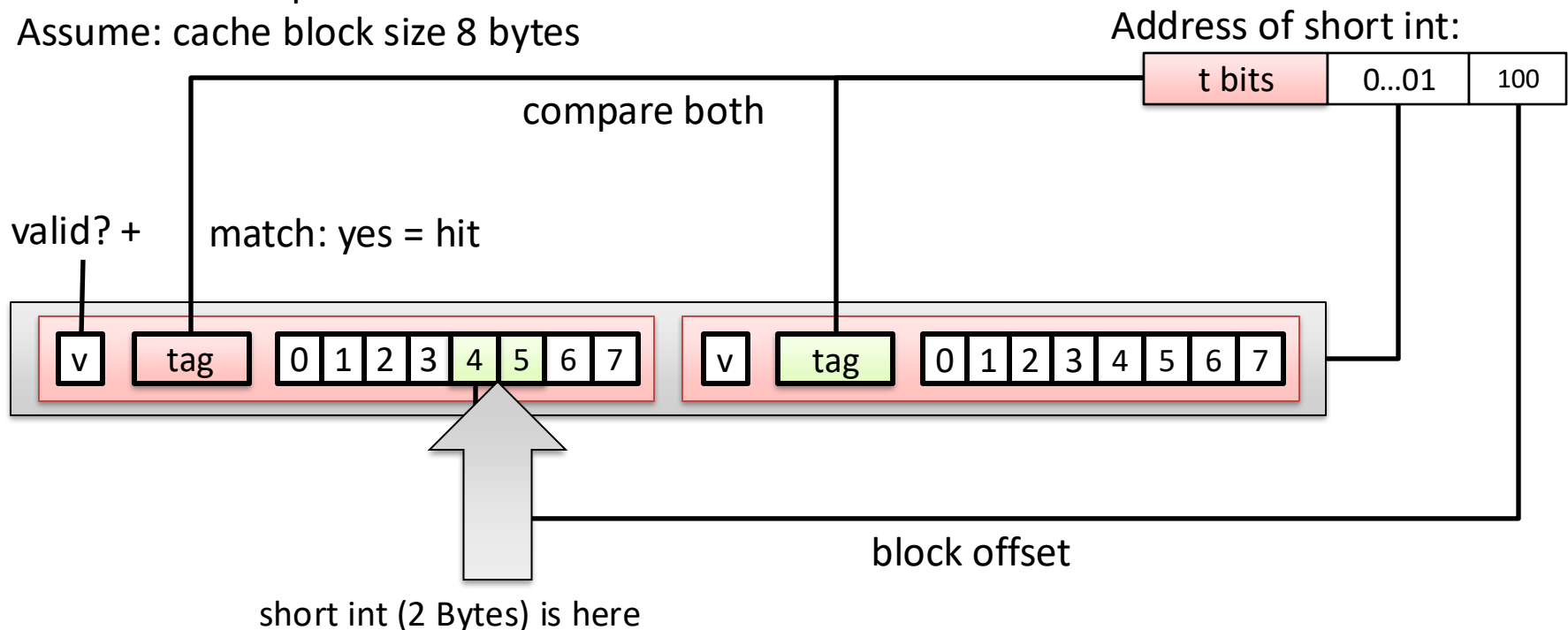
Direct mapped: One line per set
Assume: cache block size 8 bytes



2-way set-associative cache

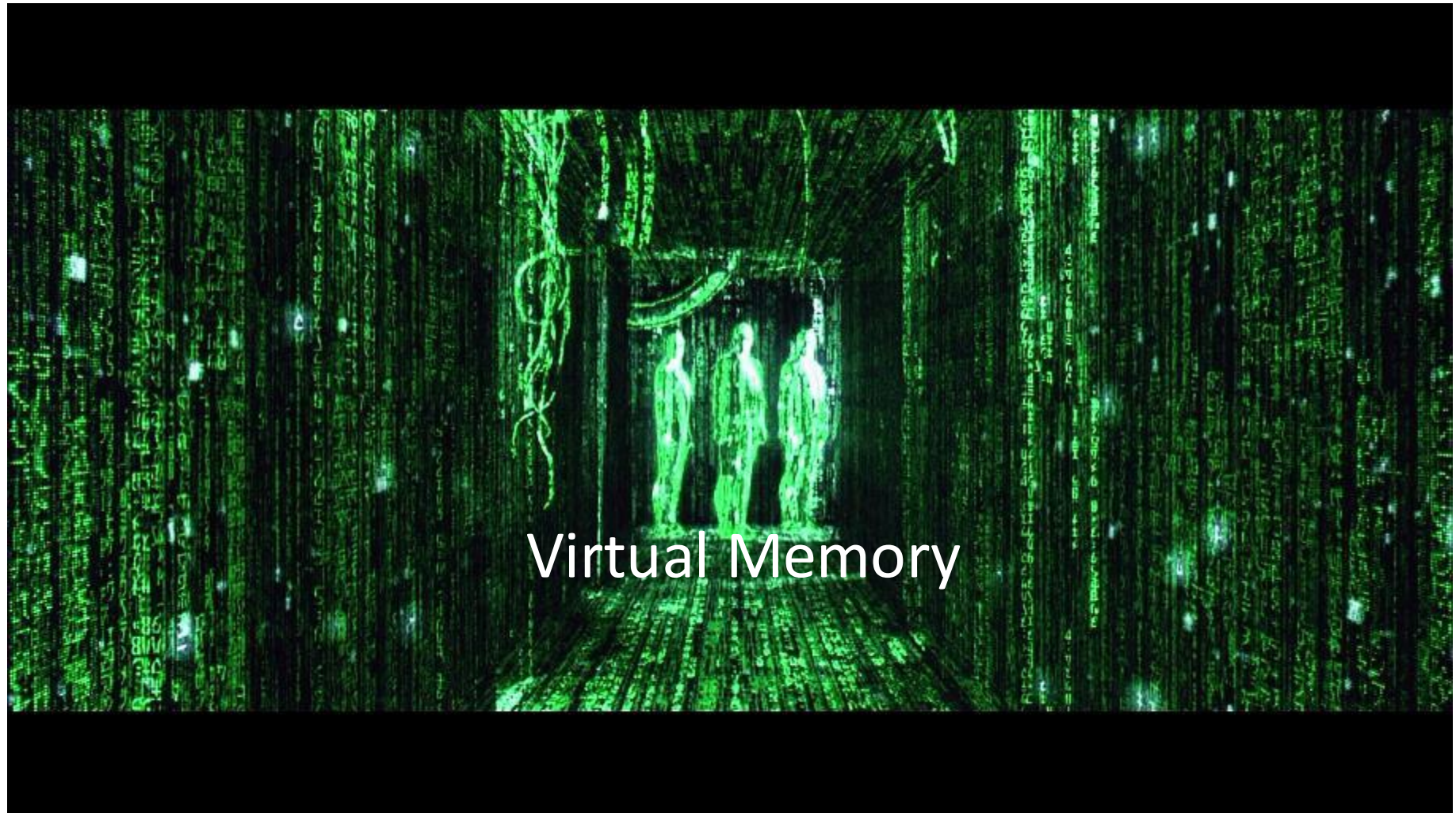
E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...



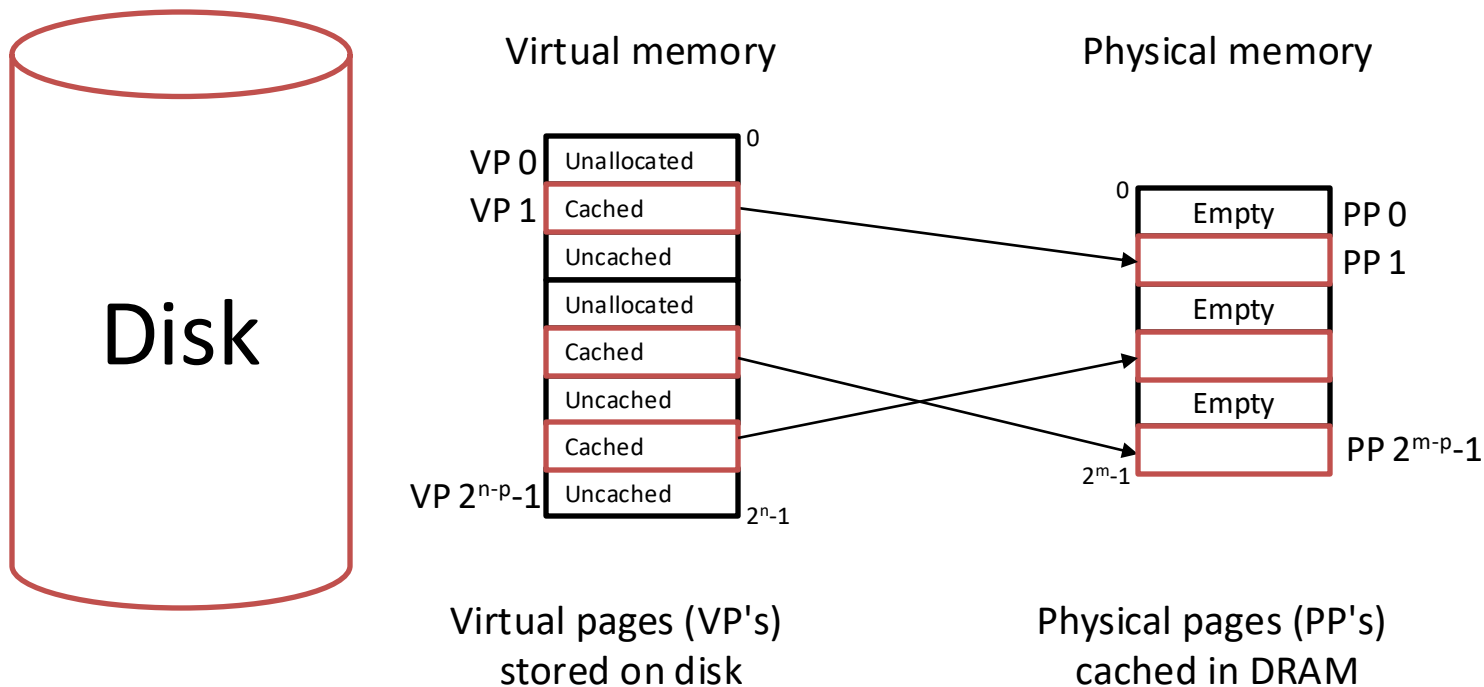
Virtual Memory: Why?

- Address Space >> Physical Memory
- Memory allocation: what goes where?
- Protection: How to restrict access
- Sharing: How to save memory

Solution: Virtual Memory and address translation!

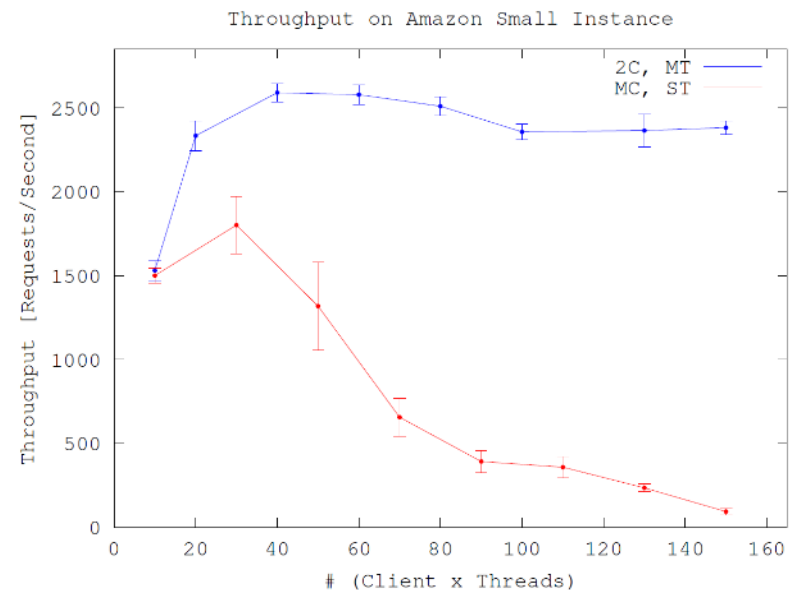
1: VM as a tool for caching

- Virtual memory: array of $N = 2^n$ contiguous bytes
 - think of the array (allocated part) as being stored on disk
- Physical main memory (DRAM) = cache for allocated virtual memory
- Blocks are called pages; size = 2^p



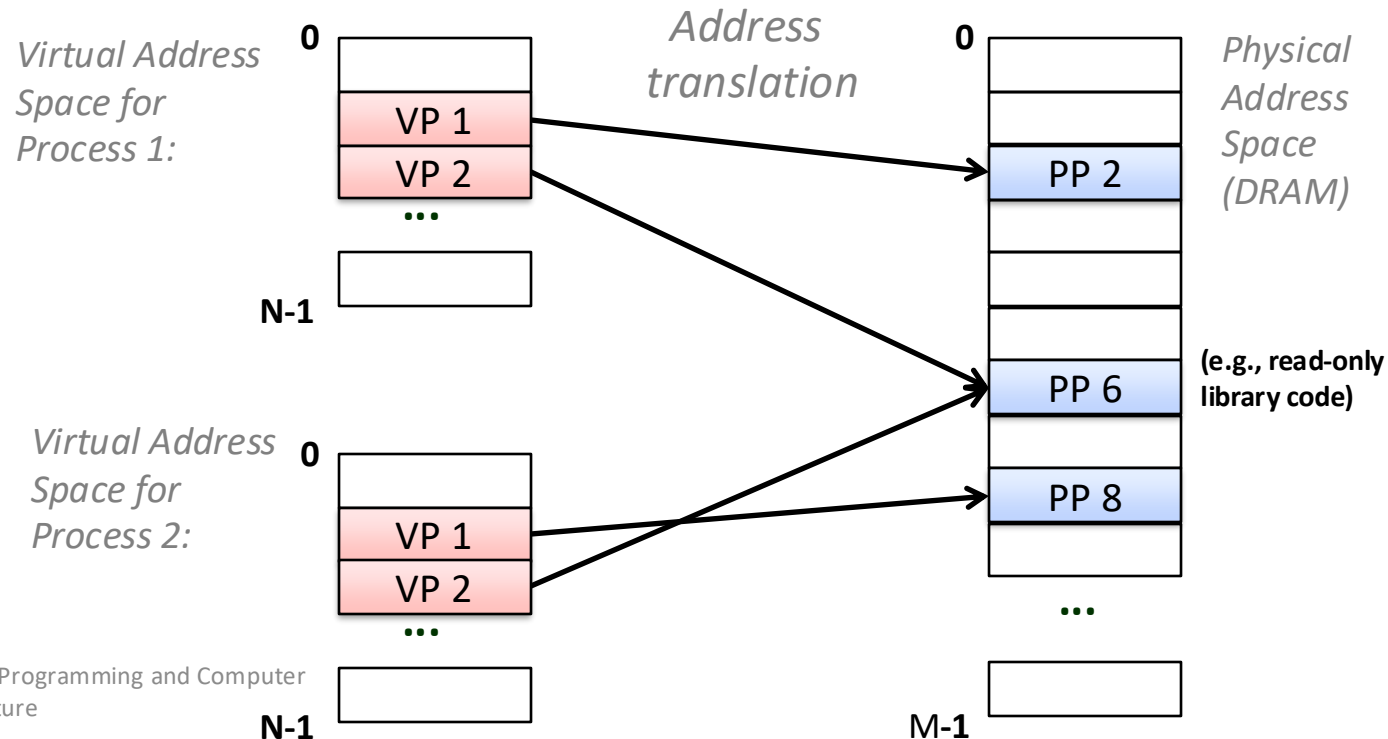
System Thrashing

- If you have a too big working set
 $\sum \text{WorkingSet} > \text{Main Memory}$
- The pages need to be swapped in and out continuously
i.e. copy from disk to memory and vice versa



2. VM as a tool for memory management

- Memory allocation
 - Each virtual page can be mapped to any physical page
 - A virtual page can be stored in different physical pages at different times
- Sharing code and data among processes
 - Map virtual pages to the same physical page (here: PP 6)



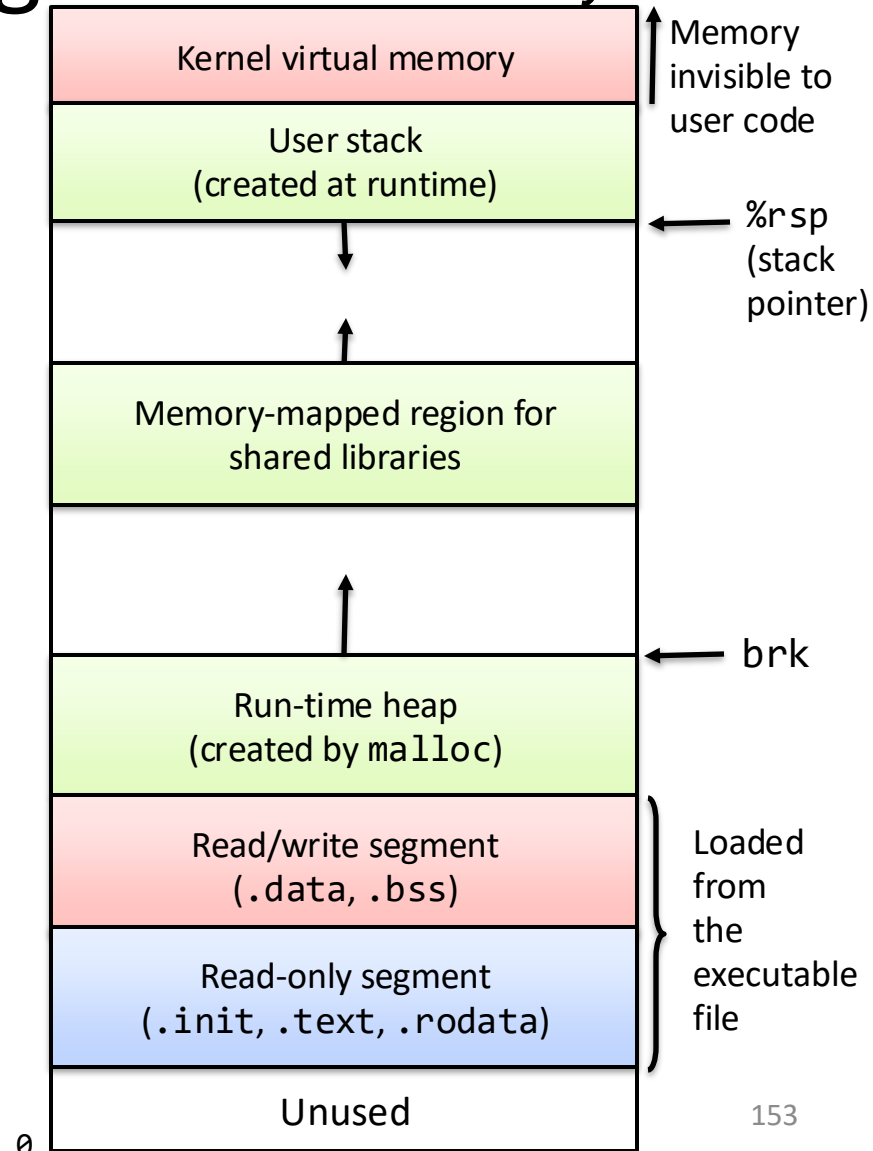
3. Using VM to simplify linking and loading

- Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

- Loading

- `execve()` allocates virtual pages for `.text` and `.data` sections = creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



4. VM as a tool for memory protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
 - If violated, send process SIGSEGV (segmentation fault)

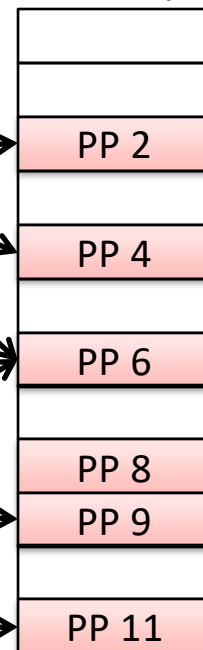
Process i:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 6
VP 1:	No	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	PP 2
⋮				

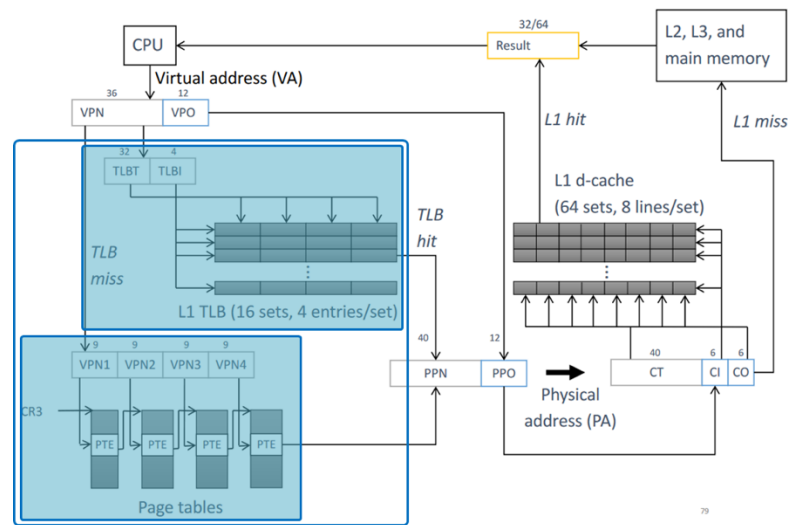
Process j:

	SUP	READ	WRITE	Address
VP 0:	No	Yes	No	PP 9
VP 1:	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	PP 11

*Physical
Address Space*



Address Translation



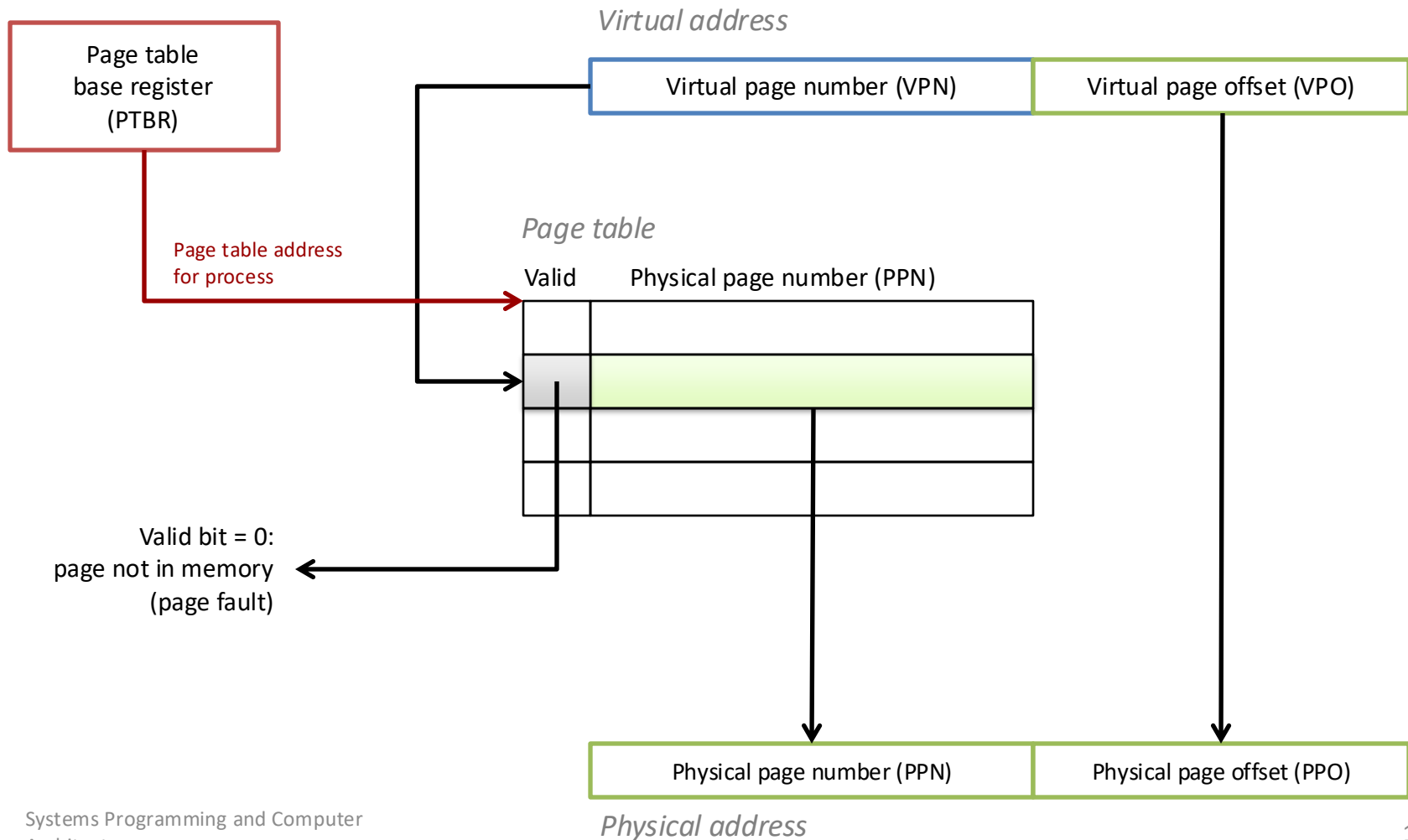
Address Translation

Address Translation

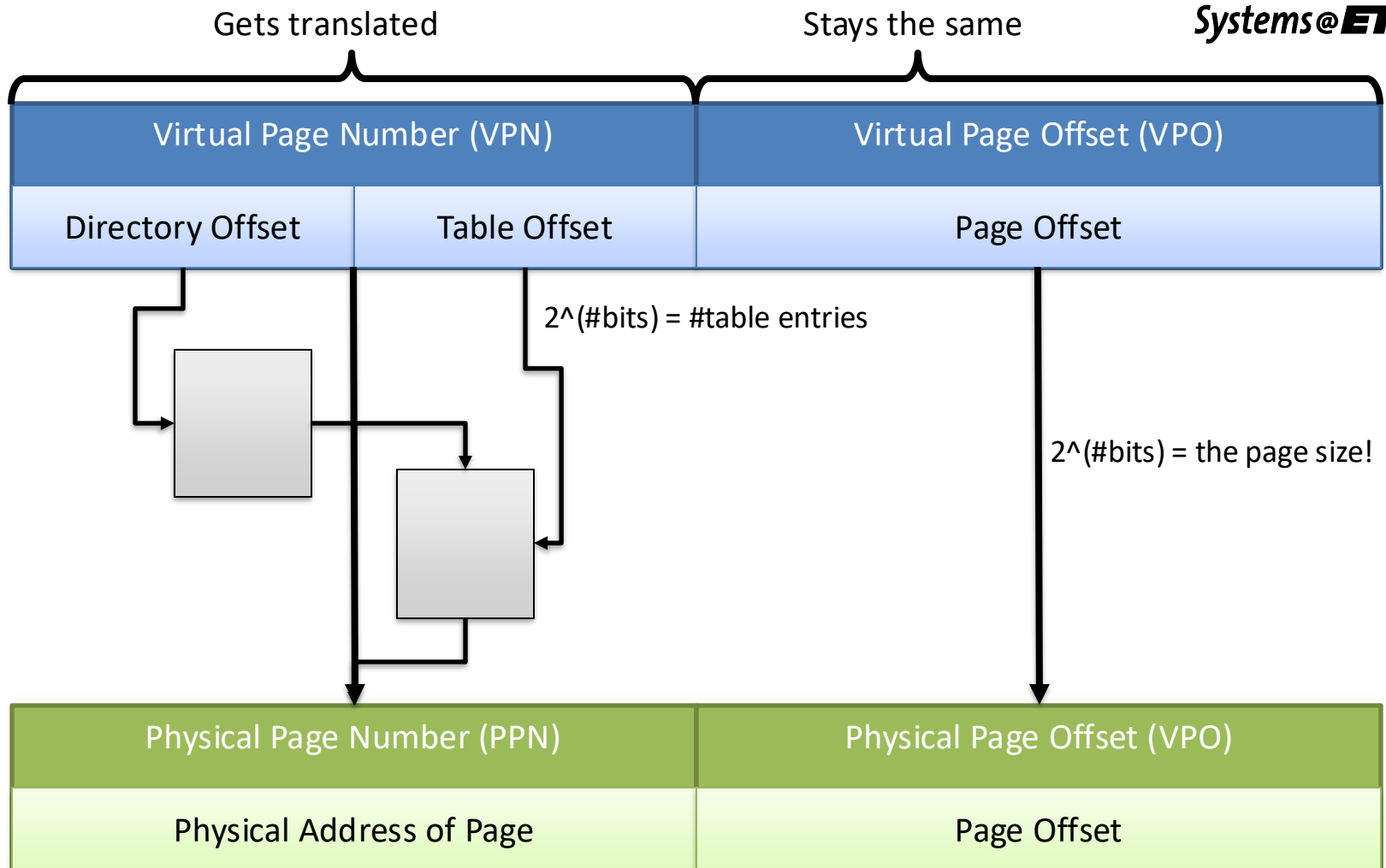


- You cannot simply store each VA->PA mapping! (too much memory usage)
- You cannot do the translation in software (too slow)
- You need a memory efficient & hardware accessible structure to store the mappings
- Concept of virtual/physical pages with page tables

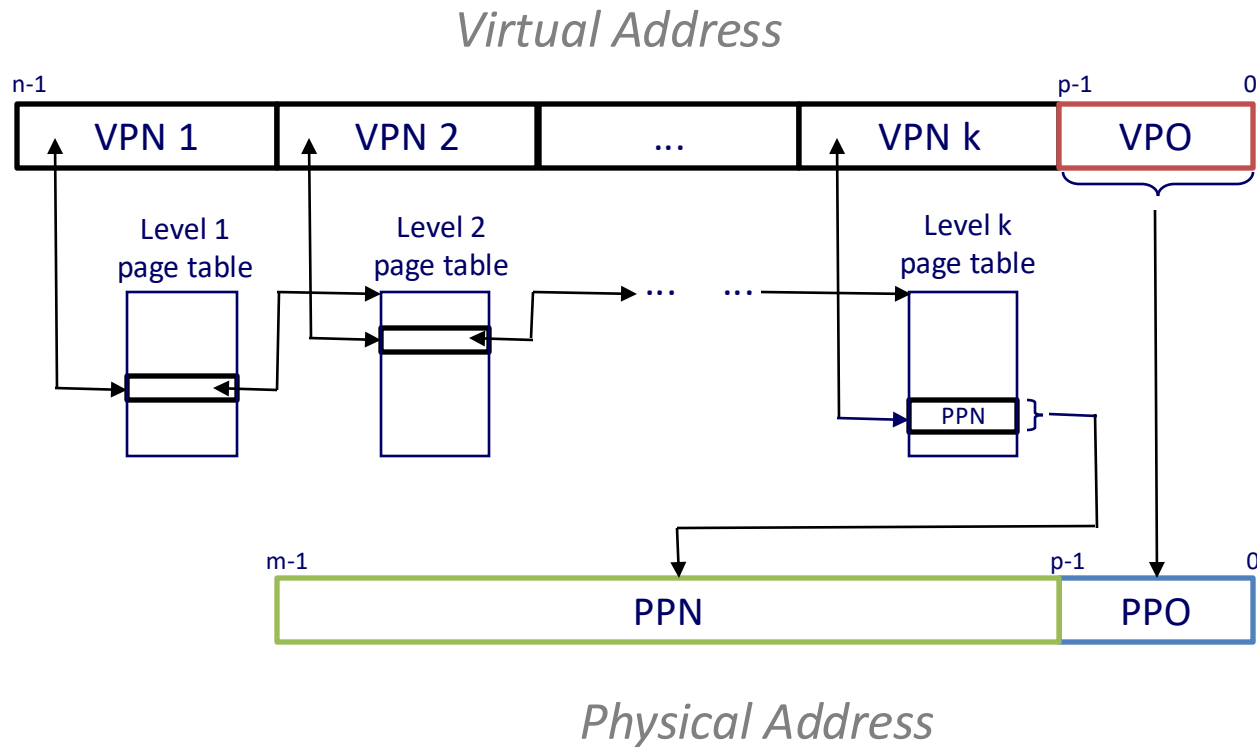
Address translation with a page table

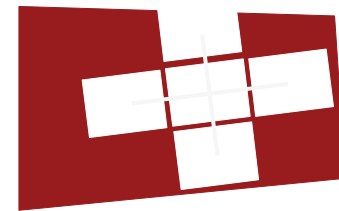


Virtual to Physical

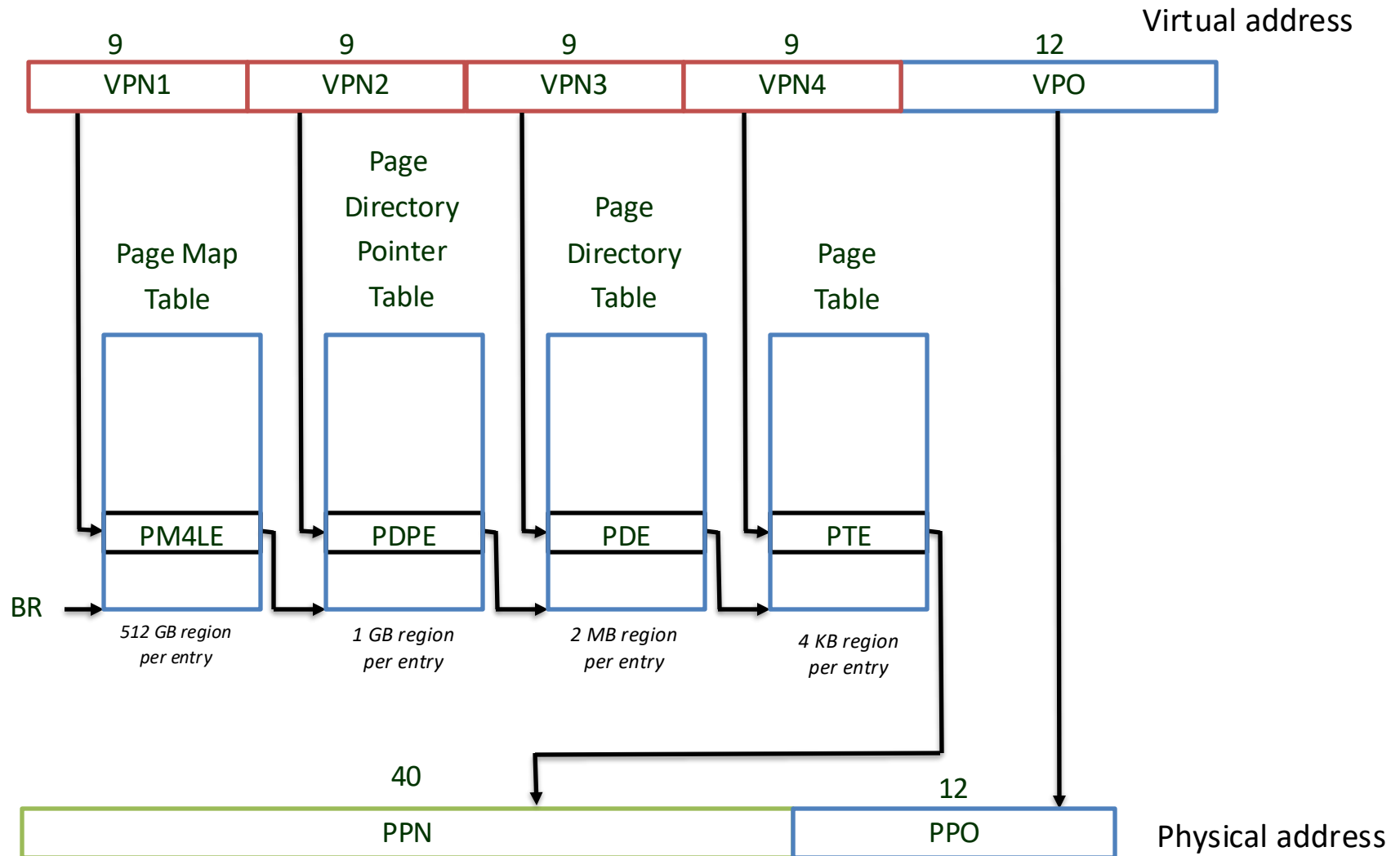


Translating with a k-level page table





x86-64 paging



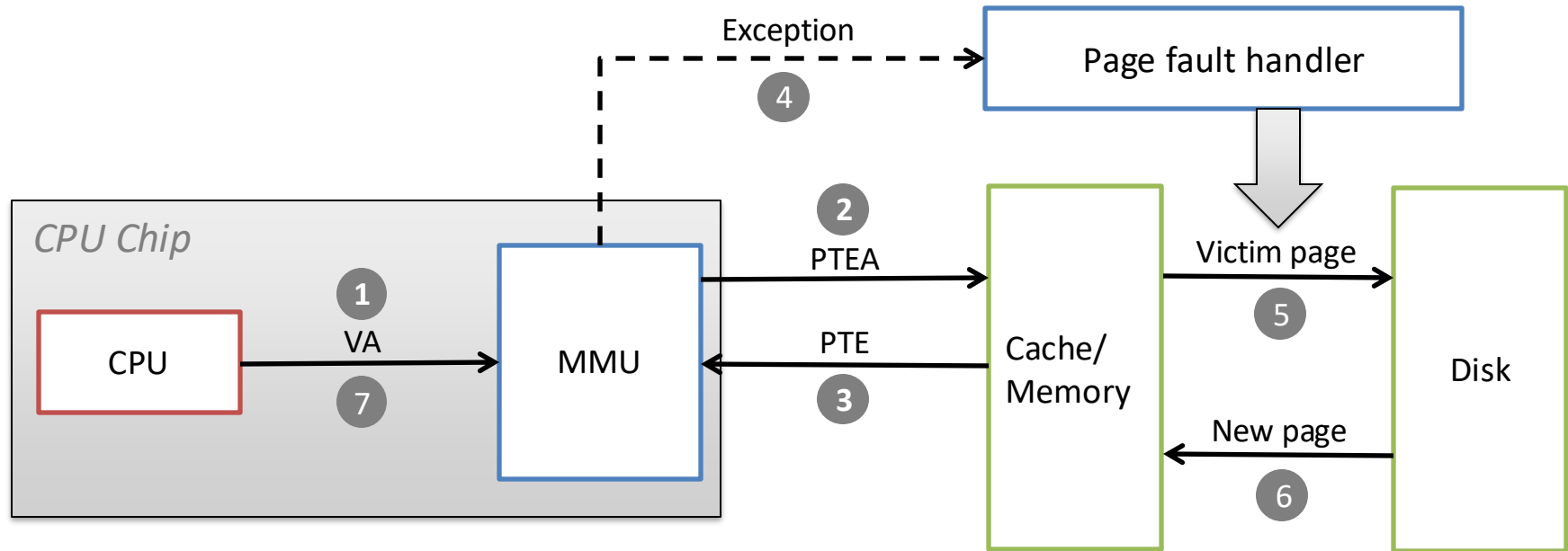
Page Tables

- The MMU walks the page table structure in hardware
- **Page hit:** successful translation, page is present in main memory
- **Page miss:** successful translation, page is not present in main memory (need to fetch from disk)

Faults vs Misses

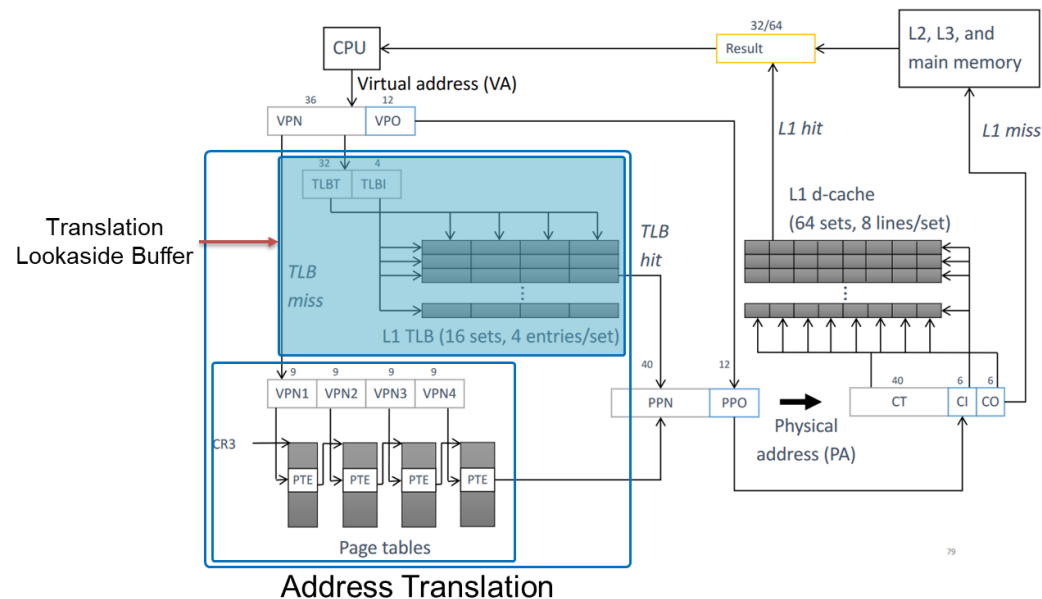
- **Page miss:** reference to virtual memory word that is not in physical memory
- **Page fault:** exception when trying to access a page
 - may be not in memory (fetch page), recoverable
 - may be not writable, error
 - may be not mapped (SIGSEGV) error

Address translation: page fault

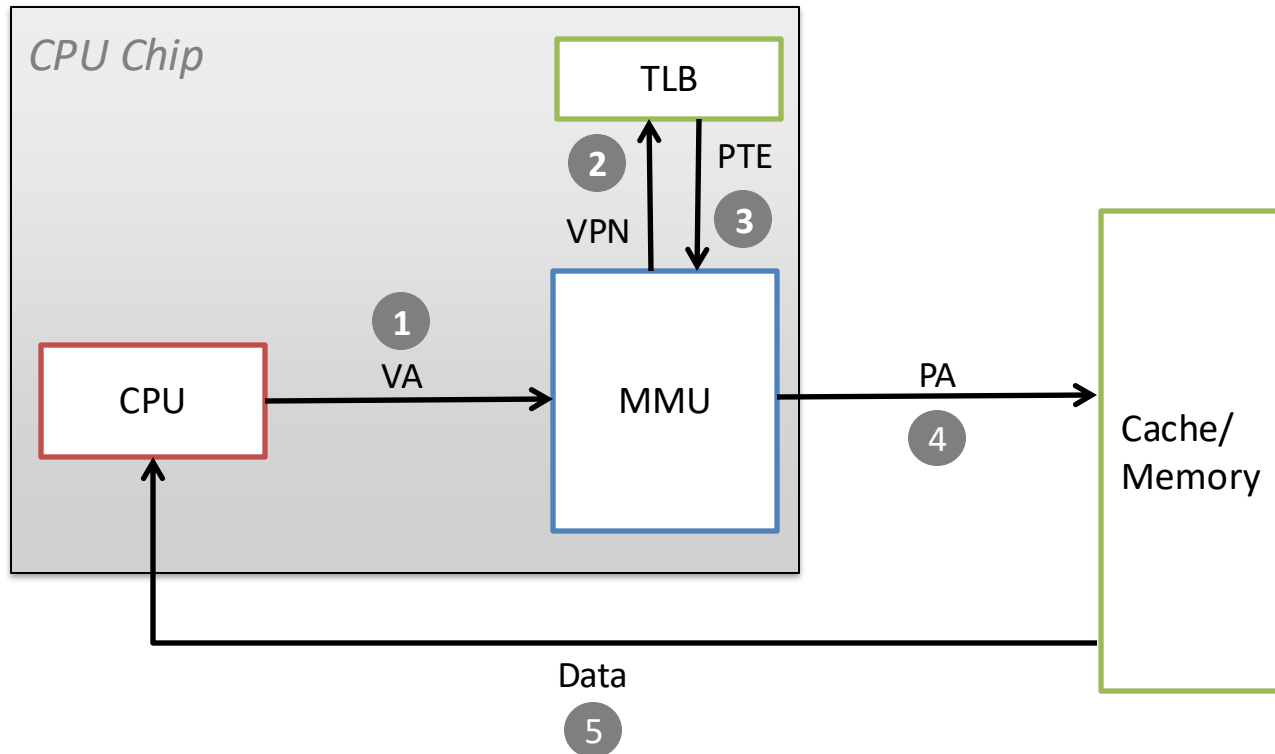


- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

Translation Lookaside Buffer (TLB)

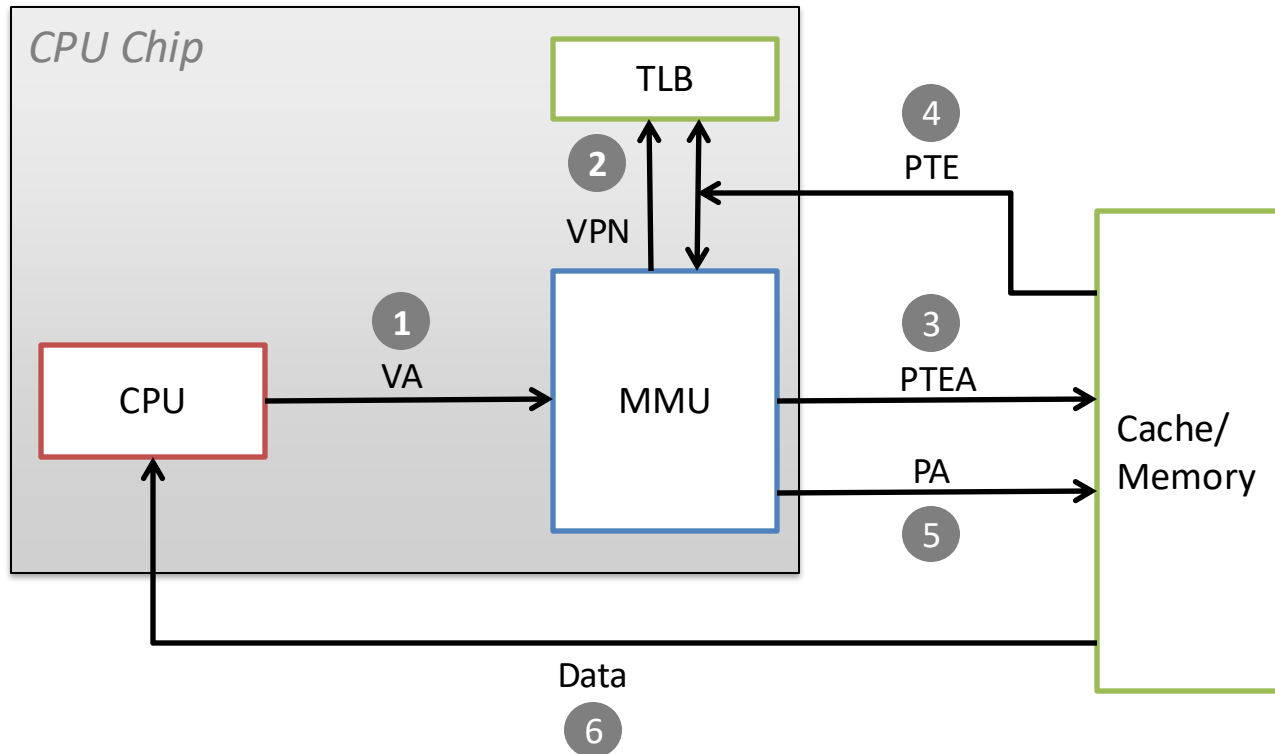


TLB hit



A TLB hit eliminates a memory access

TLB miss



A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare (we hope)



<sidenote>

- If you have a cache:
 - The cache is checked first in general
 - Then the lower-level data is checked
- When calculating the times for recovering a miss, always add the time needed to check the cache!

TLB Coverage

- Assume you have
 - 1024 entry TLB
 - 4KiB pages
 - You have 4GiB of main memory
- How much physical memory is covered by the TLB in percentage of main memory?
- How can it be increased?

TLB Coverage

- Assume you have
 - 1024 entry TLB
 - 4KiB pages
 - You have 4GiB of main me

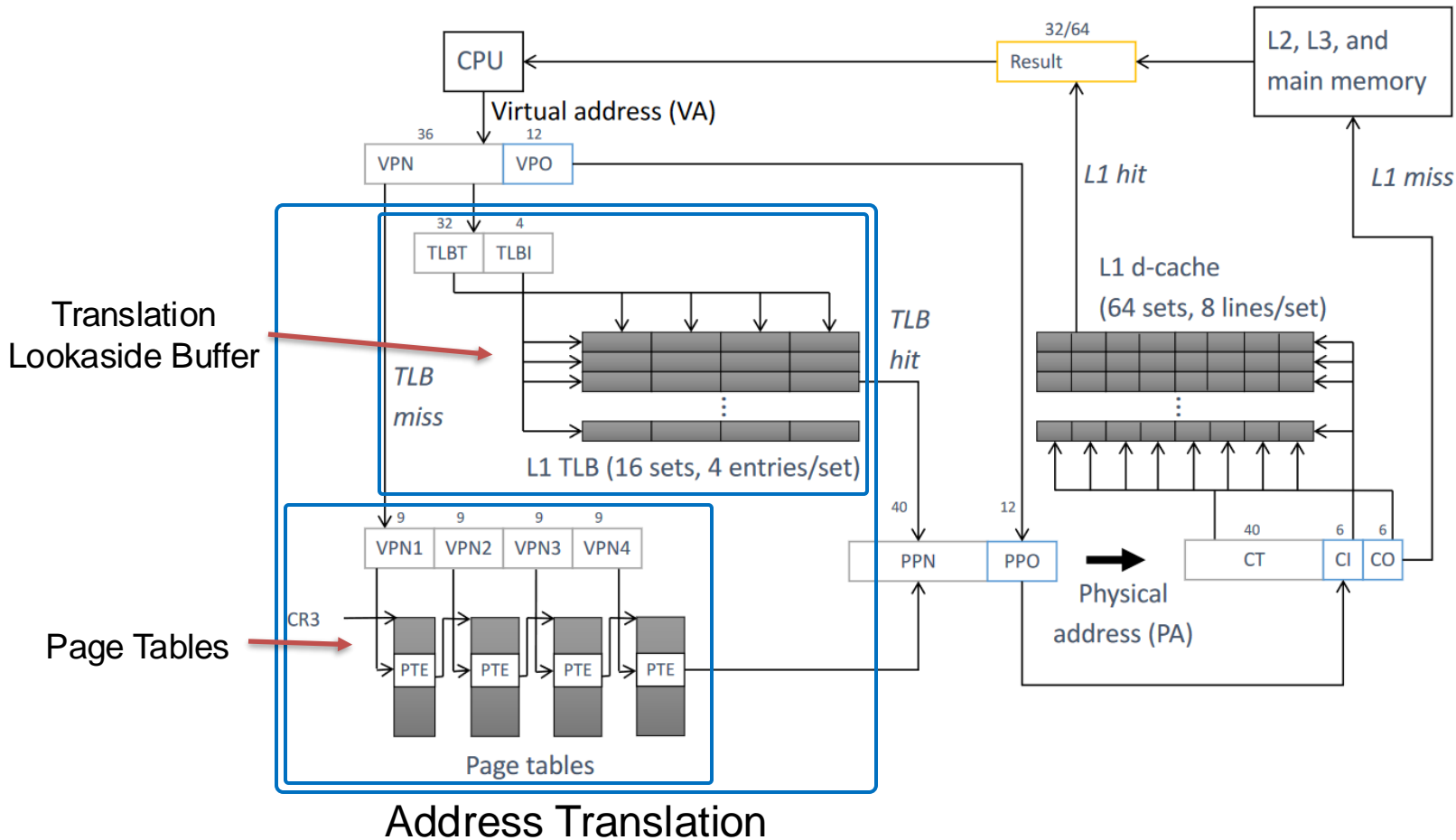
TLB covers $1024 * 4\text{KiB} = 4\text{MiB}$
 $4\text{MiB} / 4096\text{MiB} \approx 0.1\%$

- How much physical memory is covered by the TLB in percentage of main memory?

- How can it be increased?

Larger TLB (expensive)
Bigger pages

Core i7 memory system



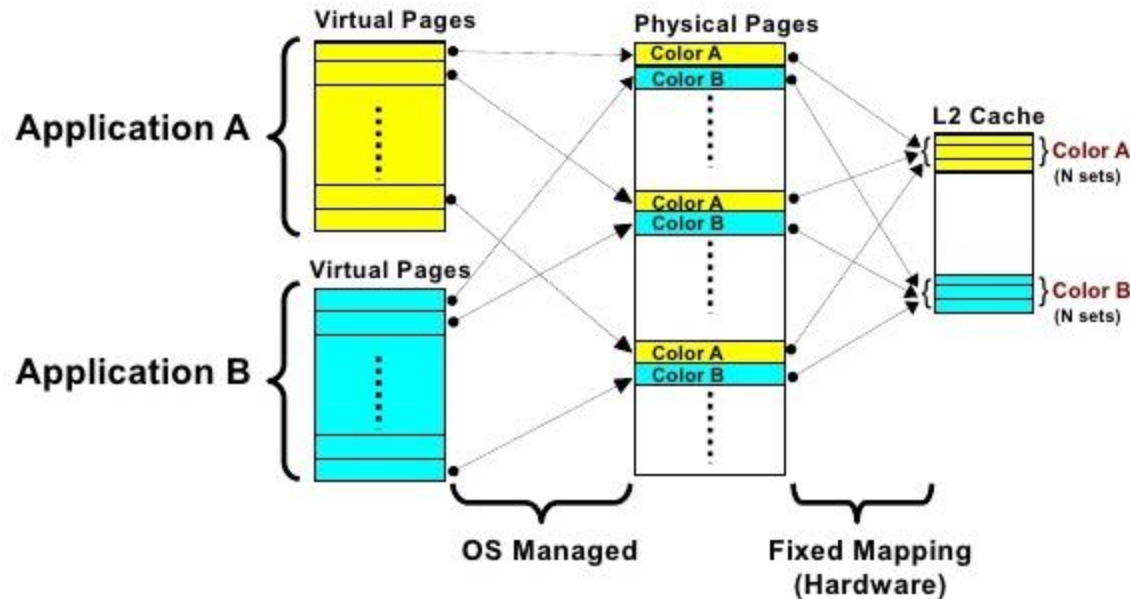
Caches and Virtual Memory

- Where to place the line is determined by its address.
 - There are [physically | virtually] tagged
[physically | virtually] indexed caches
 - Virtually is faster in general (no need to translate) but introduces aliasing (homonym and synonym problems)
http://en.wikipedia.org/wiki/CPU_cache#Address_translation

Caches and Virtual Memory

- What happens on context switch?
 - There is always the possibility that the caches have to be invalidated when another process is getting scheduled. (TLB..)
 - Two processes may interfere with each other i.e. polluting the cache resulting in a higher cache miss ratio!

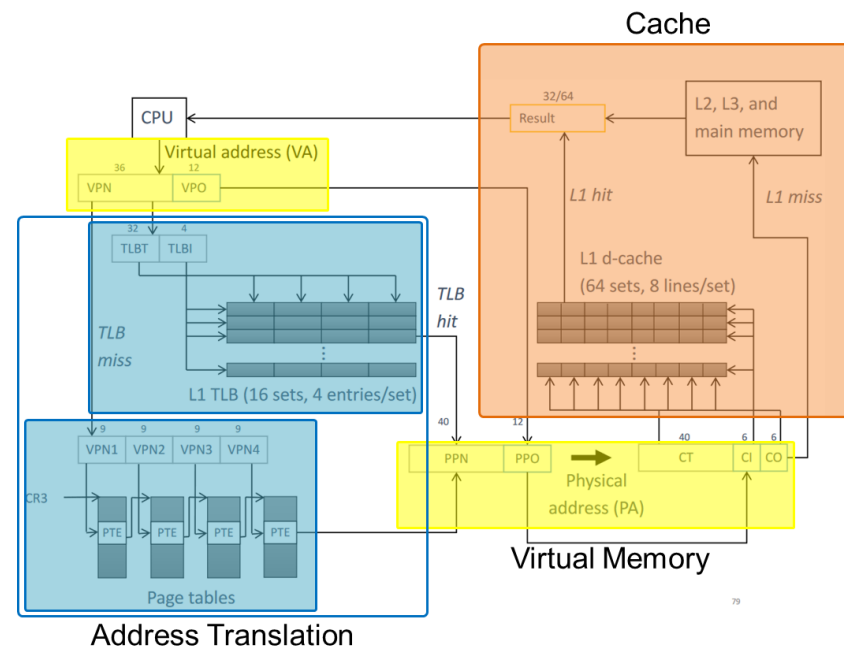
Cache coloring to restrict processes to a subset of cache?



17

What kind of cache do you need to do this?

Assignment 10



Assignment

- Pen&Paper:
 - Understand Caches, Translation/TLB
 - Cache miss rate:
$$\frac{\text{\#Cache-miss-access}}{\text{\#Total-access}}$$
- Implement a Cache simulator

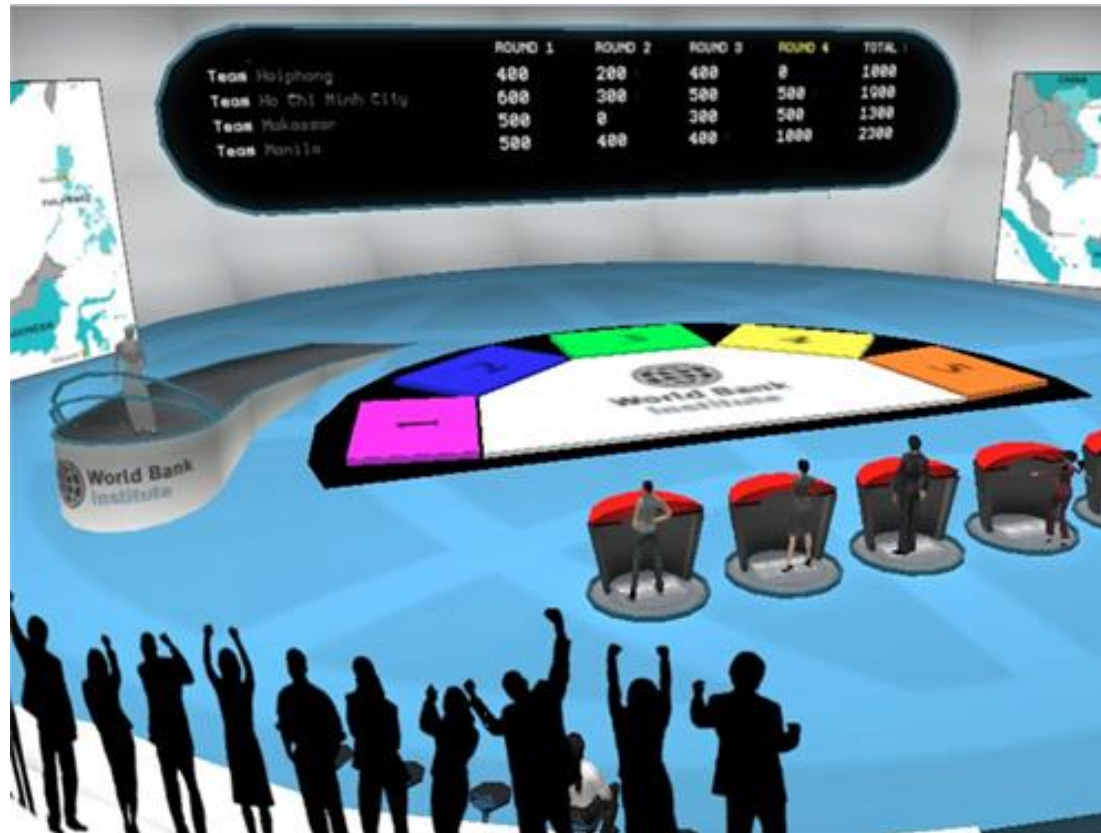
Measurement

- If you want to know how fast your program is, you will have to measure it!
- There are quite some performance counters in your CPU that gather statistics!

Example OProfile

Counter	Description
CPU_CLK_UNHALTED	Clock cycles when not halted
INST_RETIRED	number of instructions retired
LLC_MISSES	Last level cache demand requests from this core that missed the LLC
LLC_REFS	Last level cache demand requests from this core
DTLB_LOAD_MISSES	The number of DTLB load misses
L2_REQSTS	The number of Level 2 Cache requests
ICACHE_MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes Uncacheable accesses.

<http://oprofile.sourceforge.net/>



Quiz Time

Hands on Caches

Question 1

- The memory system consists of register, a single L1 cache and main memory.
- The cache is cold and the array has been initialized.
- Variables `i`, `j` and `sum` are stored in registers.
- The array `A` is aligned in memory.
- `sizeof(int) == 4`.
- The cache is direct mapped, with a block size of 8 bytes.

```
int A[2][4];

int sum()
{
    int sum = 0;

    for (int j = 0; j < 4; j++) {
        for (int i = 0; i < 2; i++) {
            sum += A[i][j];
        }
    }
    return sum;
}
```

Question 1a

- a) Suppose that the cache consists of 2 sets.
Fill out the table to indicate if the corresponding memory access in A will be a hit (**h**) or a miss (**m**).

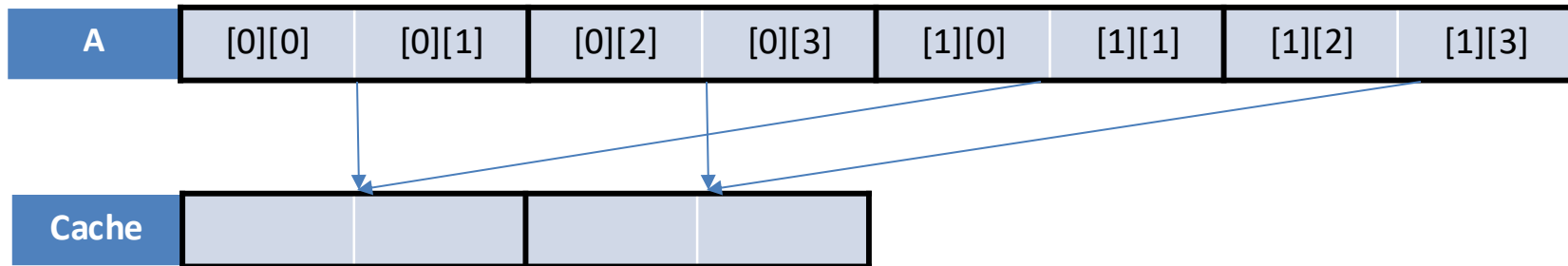
A	Col 0	Col 1	Col 2	Col 3
Row 0	M			
Row 1				

Question 1a

2 sets -> 1 bit



blocksize = 8 -> 3 bits offset



A	Col 0	Col 1	Col 2	Col 3
Row 0	M	M	M	M
Row 1	M	M	M	M

Question 1b

- a) Suppose that the cache consists of 2 sets.
Fill out the table to indicate if the corresponding memory access in A will be a hit (**h**) or a miss (**m**).
- b) What is the pattern of hits and misses if the cache consists of 4 sets instead of 2 sets?

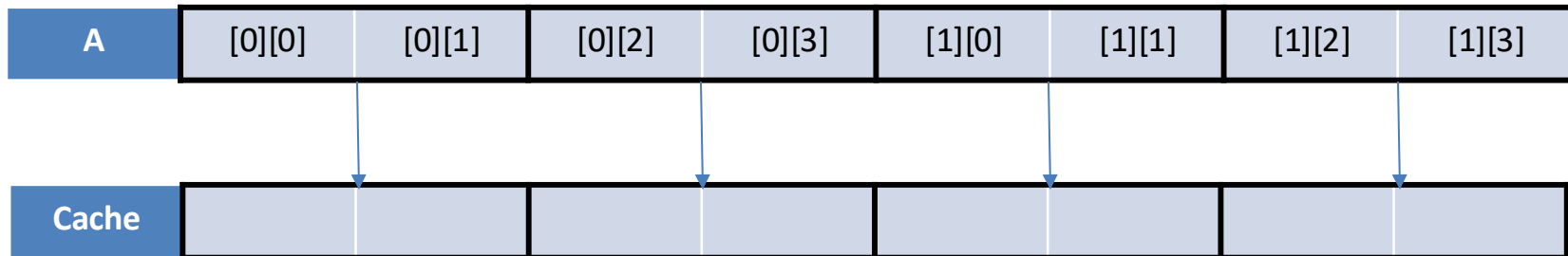
A	Col 0	Col 1	Col 2	Col 3
Row 0	M			
Row 1				

Question 1b

4 sets -> 2 bit



blocksize = 8 -> 3 bits offset



A	Col 0	Col 1	Col 2	Col 3
Row 0	M	H	M	H
Row 1	M	H	M	H

Question 2

- `sizeof(int) == 4`.
- Array `x` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `x`.
- All variables are stored in the registers.

```
int x[128];  
int j;  
int sum = 0;  
  
for (int i = 0; i < 64; i++) {  
    j = i + 64;  
    sum += x[i] * x[j];  
}
```

Question 2.1

a) What is the cache miss rate?

$$256 = \text{blocksize} \times \text{\#sets} \times \text{\#ways} \quad \text{32 sets}$$

32 sets \rightarrow 5 bit

Address	8	7	6	5	4	3	2	1	0
---------	---	---	---	---	---	---	---	---	---

blocksize = 8 \rightarrow 3 bits offset

$$\&x[j] = \&x[i] + 64 \times 4 = \&x[i] + 256 \quad \text{always map to the same block: 100\% missrate}$$

b) If the cache were twice as big, what would be the miss rate?

$$512 = \text{blocksize} \times \text{\#sets} \times \text{\#ways} \quad \text{64 sets}$$

64 sets \rightarrow 6 bit

Address	8	7	6	5	4	3	2	1	0
---------	---	---	---	---	---	---	---	---	---

blocksize = 8 \rightarrow 3 bits offset

2 elements / block: Every 2nd iteration is a hit, every other is a miss: 50% miss-rate

Question 2.2

256 = blocksize x #sets x #ways
LRU Policy

📖👤 16 sets, 2 ways, 8 bytes

16 sets -> 4 bit

Address	8	7	6	5	4	3	2	1	0
---------	---	---	---	---	---	---	---	---	---

blocksize = 8 -> 3 bits offset

$\&x[j] = \&x[i] + 64 \times 4 = \&x[i] + 256$

📖👤 always map to the same set.

📖👤 we have 2 blocks / set

📖👤 50% missrate

Larger cache-size does not help!

2 elements / block: Every 2nd element is a hit, every other is a miss: 50% miss-rate

Larger cache line size does help!

Miss-rate = $1/\text{\#elements per block}$

Questions?

