

SoC 101:

a.k.a., “*Everything you wanted to know about a computer but were afraid to ask*”


Lecture 6: The Memory Hierarchy

Prof. Adam Teman
EnICS Labs, Bar-Ilan University


14 June 2023




Lecture Overview



Introduction to the Memory Hierarchy

 Emerging Nanoscale Integrated Circuits and Systems Lab


The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 




Cache Organization


 Emerging Nanoscale Integrated Circuits and Systems Lab

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 



Tradeoffs in Cache Design


 Emerging Nanoscale Integrated Circuits and Systems Lab

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 



Virtual Memory

 Emerging Nanoscale Integrated Circuits and Systems Lab


The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 




Practical Paging


 Emerging Nanoscale Integrated Circuits and Systems Lab

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 



The Translation Lookaside Buffer (TLB)

 Emerging Nanoscale Integrated Circuits and Systems Lab

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University 

Memory
Hierarchy

Cache
Organization

Design
Tradeoffs

Virtual
Memory

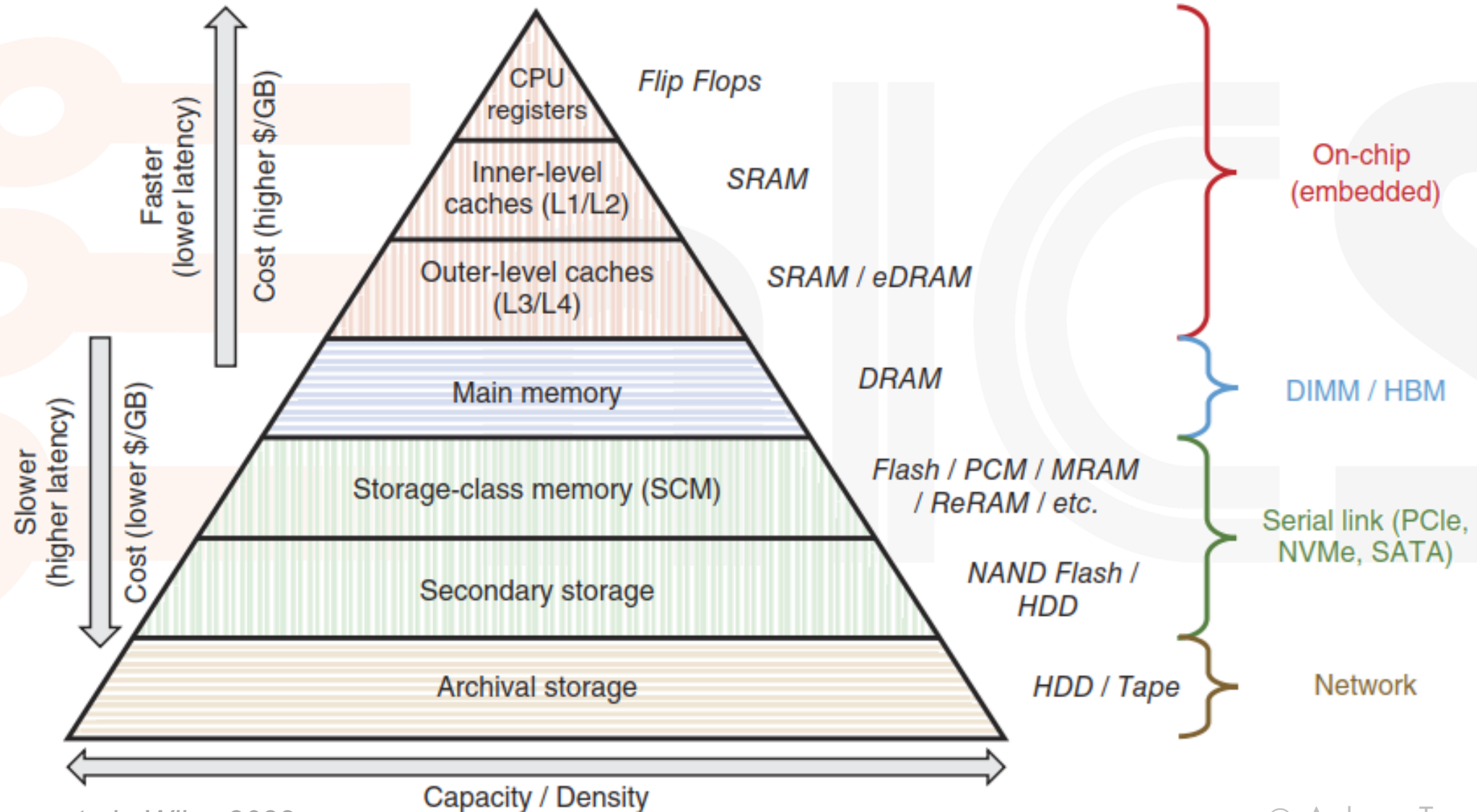
Practical
Paging

TLB

Introduction to the Memory Hierarchy



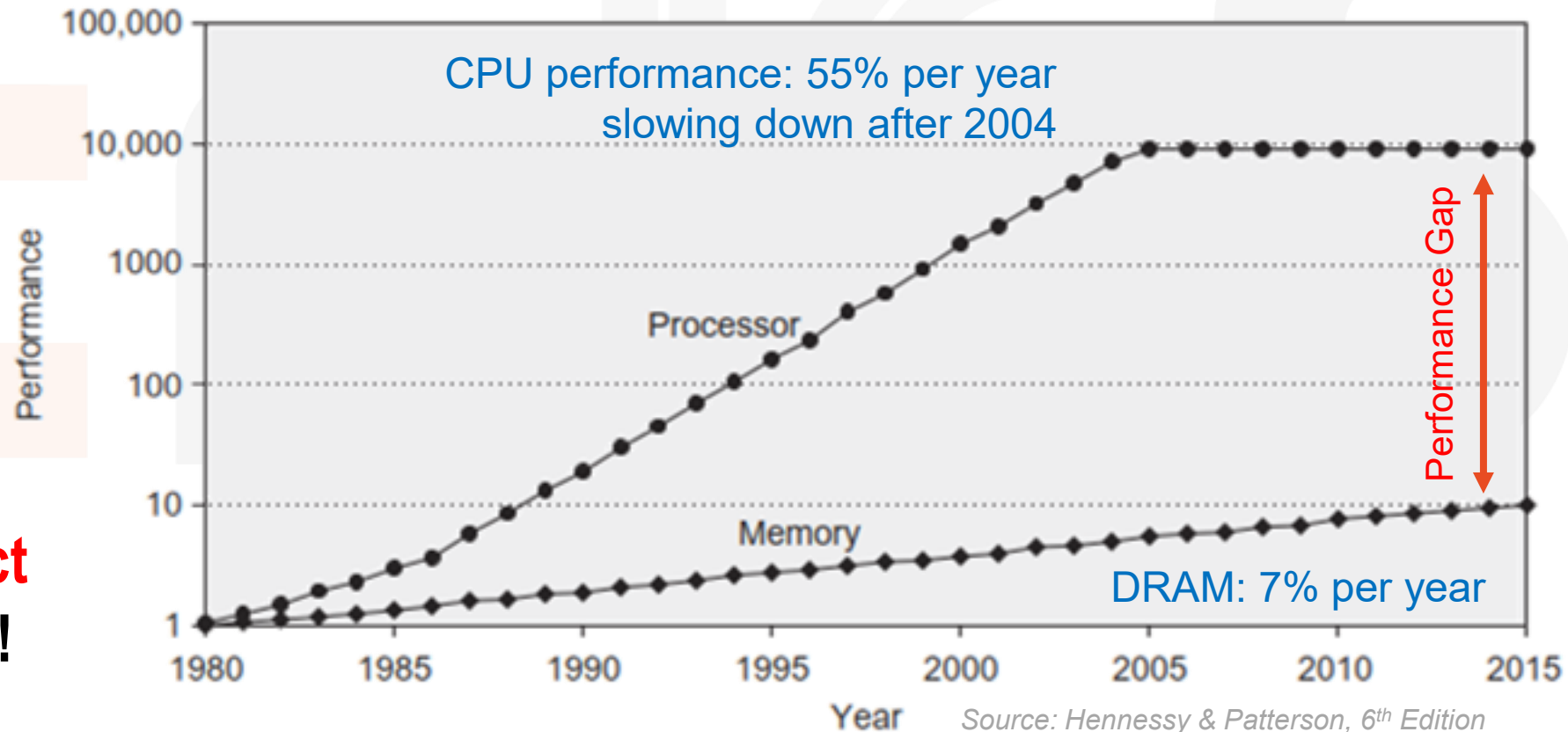
Reminder: The Memory Hierarchy



Processor-DRAM Gap (Latency)

- 1980 microprocessor executes **~one instruction** in same time as DRAM access
- 2017 microprocessor executes **~1000 instructions** in same time as DRAM access

- Slow DRAM access has **disastrous impact** on CPU performance!



Typical Memory Access Patterns

- **Instruction Fetches**

- Sequential – consecutive locations
- Code loops – repetitive locations
- Branches and subroutine calls

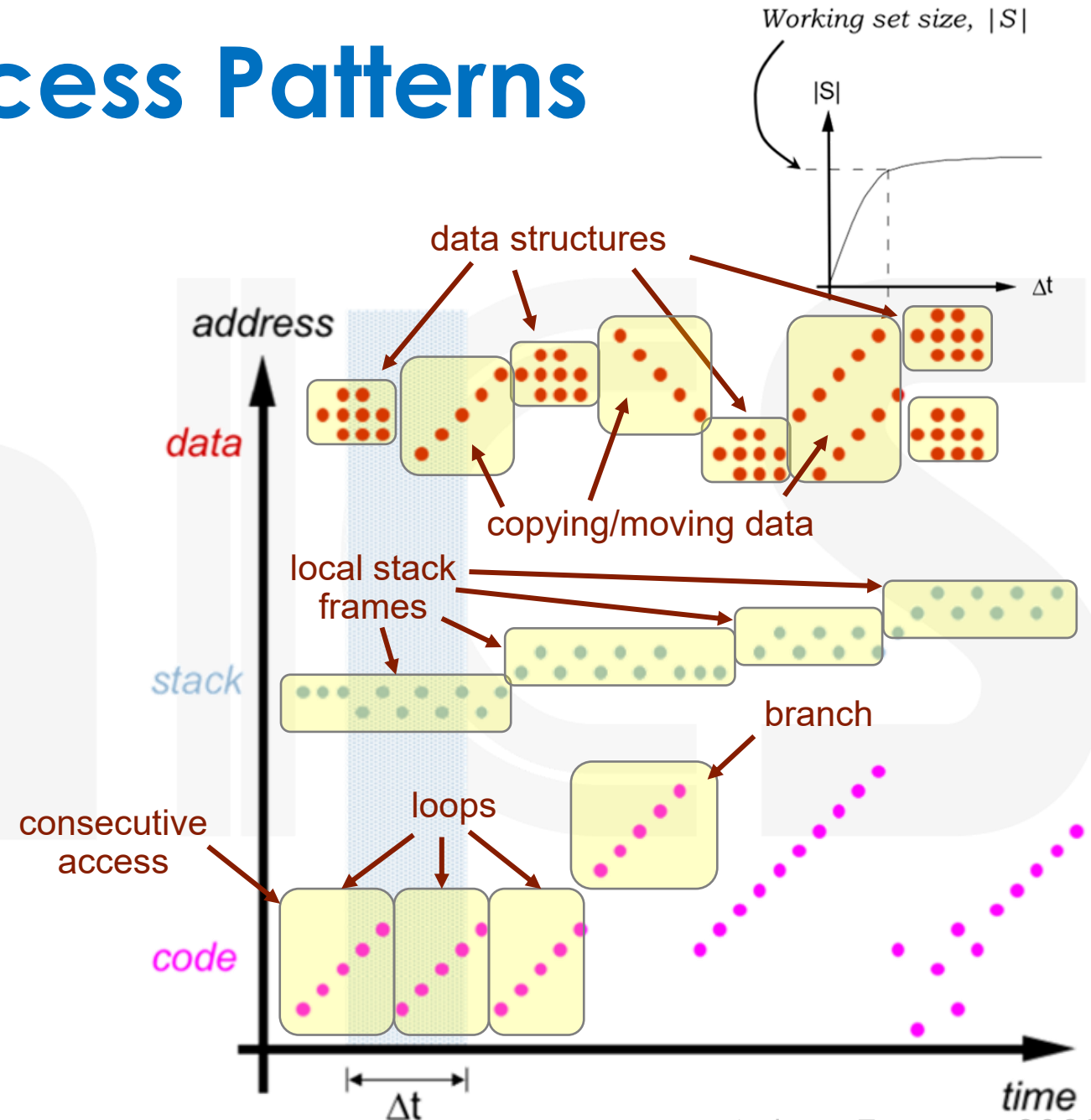
- **Stack Frame**

- Local to small region of memory

- **Data Access**

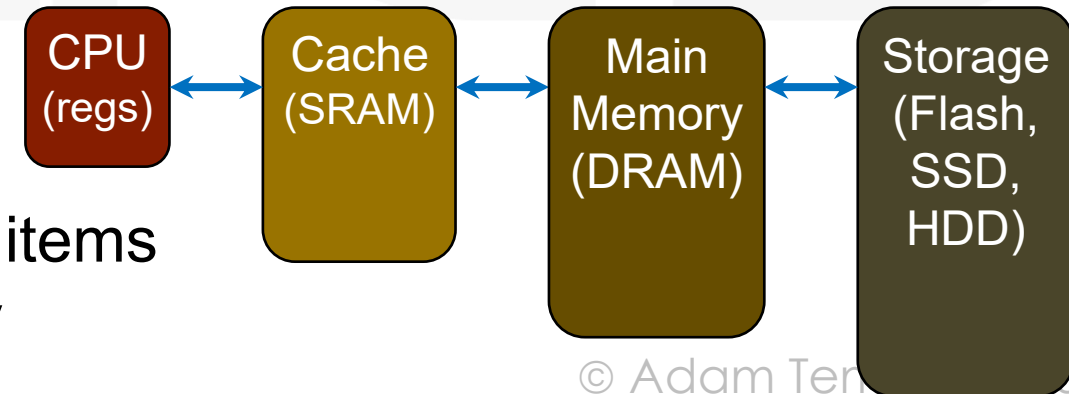
- Object or struct components
- Elements of an array
- Movement between data structures

- **Limited working set size**




The Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality (*locality in time*)
 - Items accessed **recently** are likely to be accessed again soon
 - e.g., instructions in a loop, stack variables, accessed data structures
- Spatial locality (*locality in space*)
 - Items **near** those accessed recently are likely to be accessed soon
 - E.g., sequential instruction access, array/struct data
- Taking advantage of locality
 - Store everything on disk
 - Copy recently accessed (and nearby) items from disk to smaller DRAM memory
 - Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory



Memory Caching

- Mismatch between processor and memory speeds leads to a new level:
a memory cache

 **cache**
/kaSH/

noun

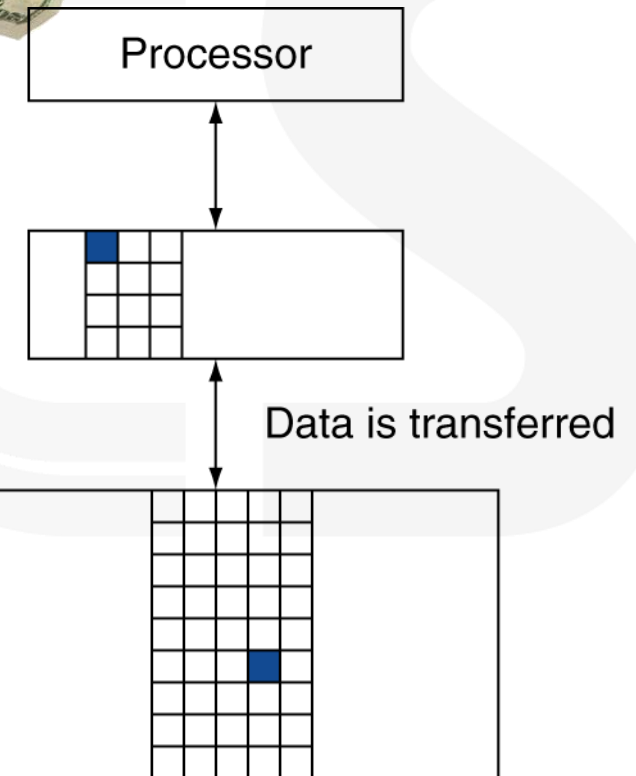
1. a collection of items of the same type stored in a hidden or inaccessible place.
"an arms cache"
synonyms: hoard, store, stockpile, stock, supply, collection, accumulation, reserve, fund;

verb

1. store away in hiding or for future use.

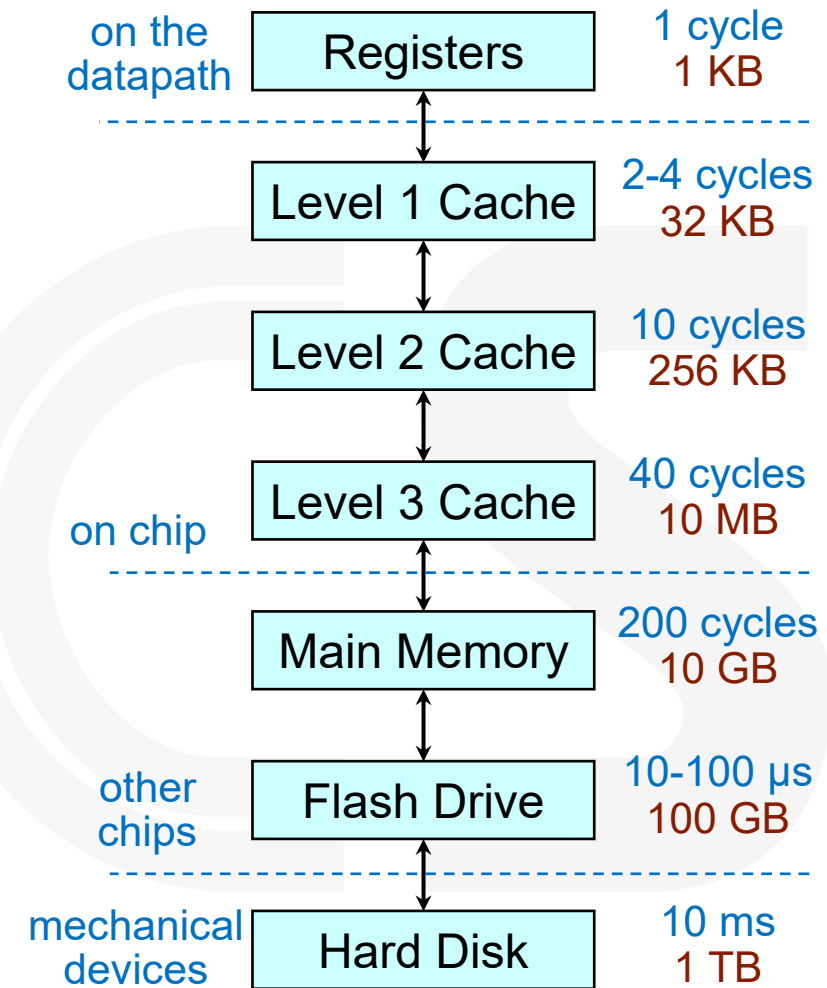
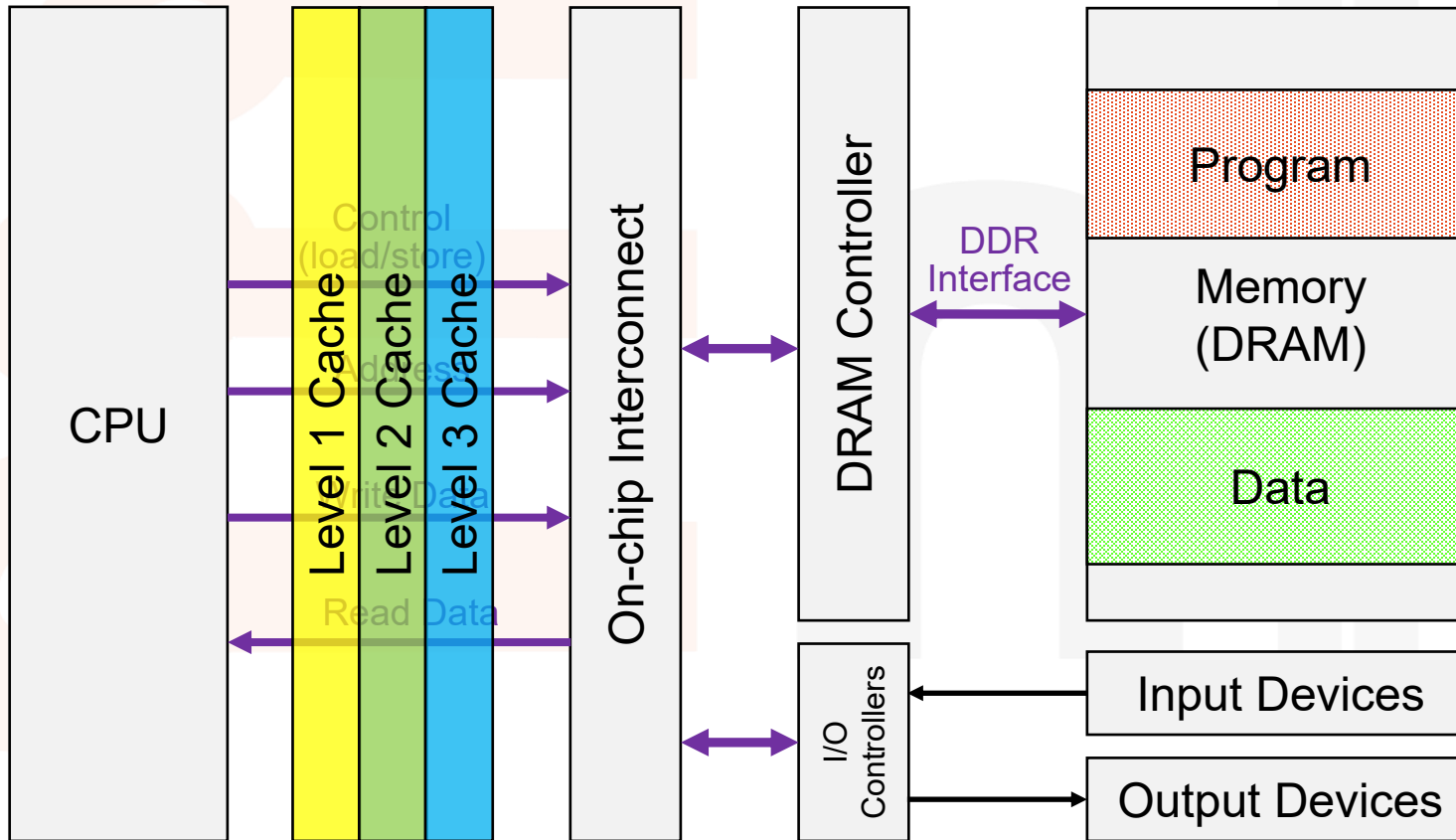


- Implemented with **same IC processing technology** as the CPU (usually integrated on same chip)
 - **Faster** but **more expensive** than DRAM memory.
- Cache is a copy of a **subset** of main memory.



Source: Patterson & Hennessy

Adding Cache to Computer



The memory hierarchy is **Inclusive**: what is in **L1\$** is a **subset** what is in **Main Memory** that is a **subset** of is in **Secondary Memory**



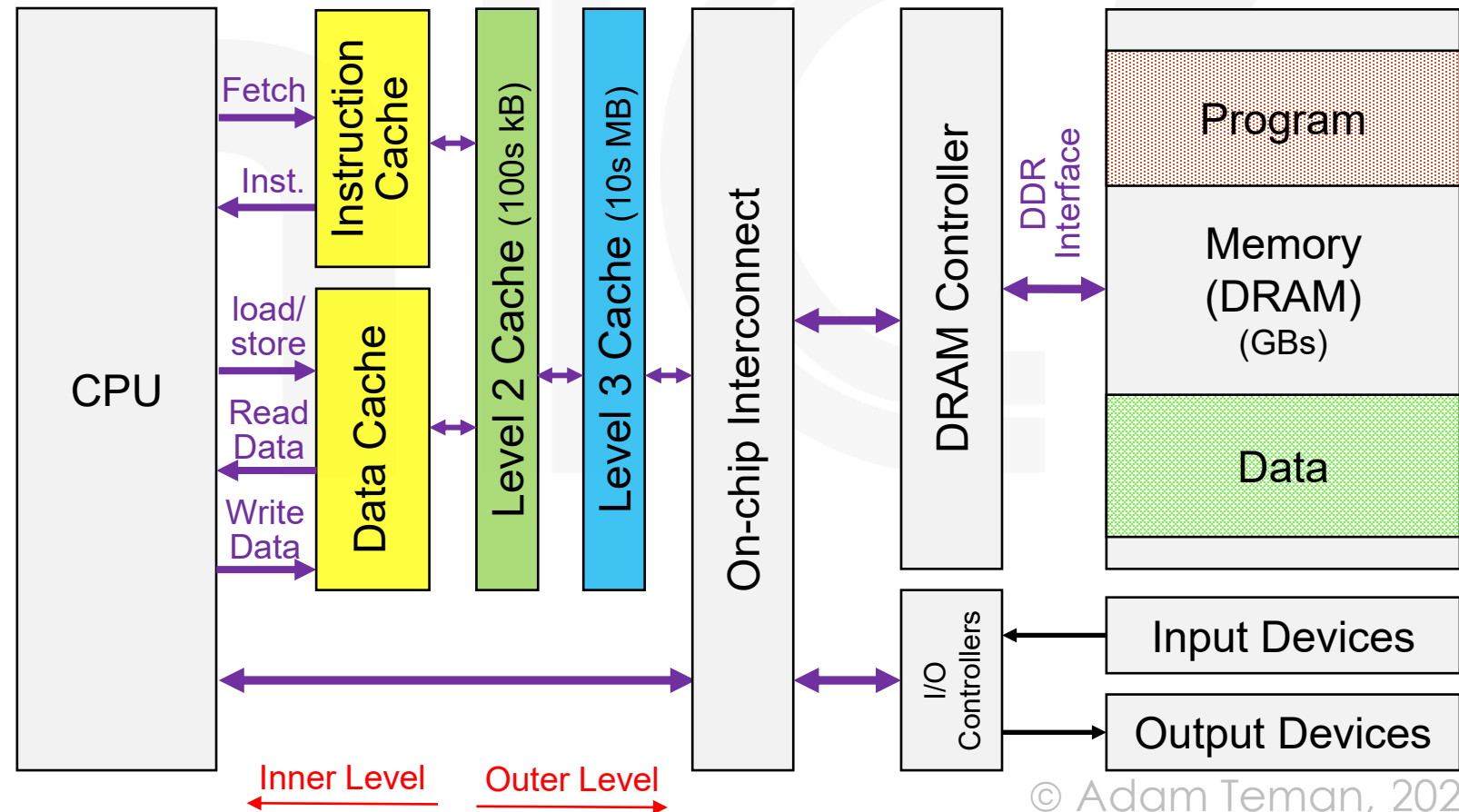
Cache Organization



Reminder: Adding Cache to Computer

- A cache exploits **data locality** to reduce the number of external memory accesses.
- **The idea:** Copy commonly accessed data to an on-chip memory (SRAM)

- Inner level cache (L1) holds a copy of outer levels (L2, DRAM).
- L1 is often divided into instruction and data cache, while outer levels are shared.



A basic cache

- A cache is a subset (copy) of an outer level of memory
 - When accessing data, we need to check if it is in the cache.
 - Therefore, every byte of data needs to be tagged with an ID.

```
lw t0, ID 3  
lw t1, ID X
```

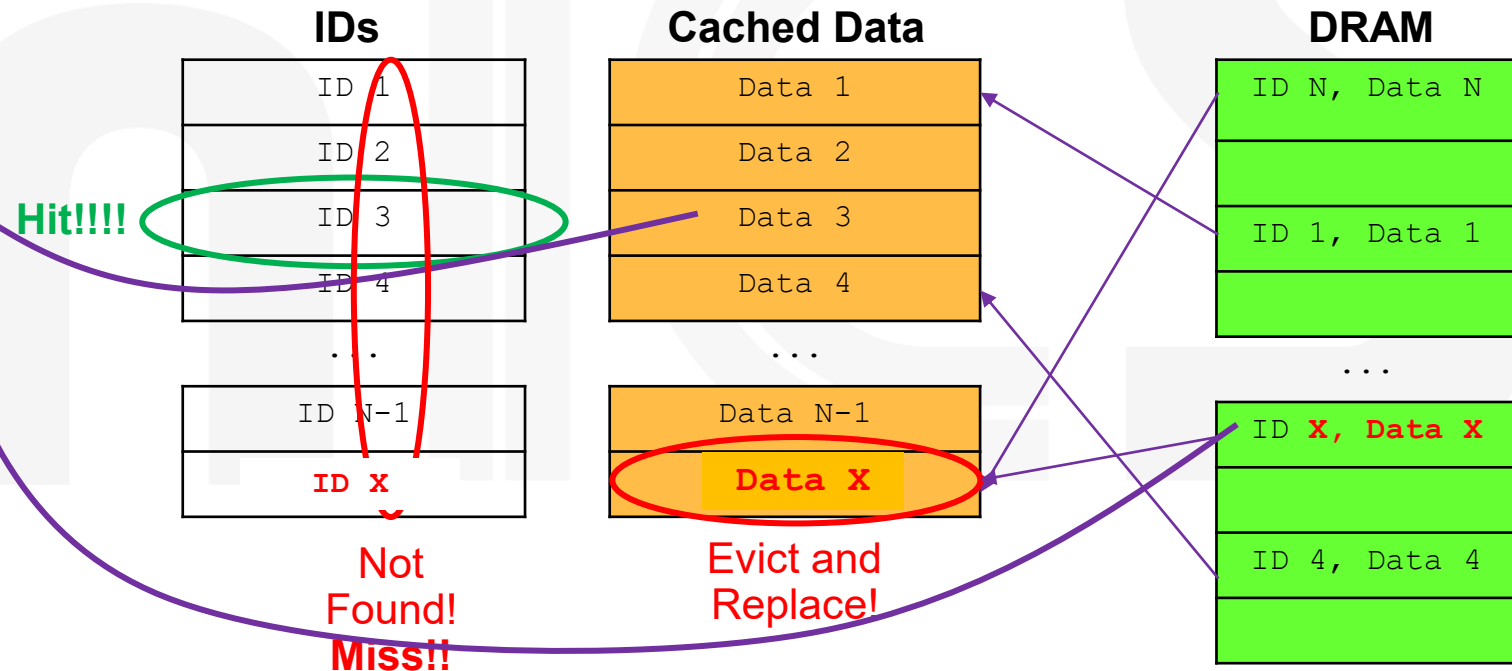
t0	Data 3
t1	Data X

Time elapsed = HitTime

Time elapsed = MissPenalty

Hit Ratio: $HR = \frac{hits}{hits+misses} = 1 - MR$

Miss Ratio: $MR = \frac{misses}{hits+misses} = 1 - HR$

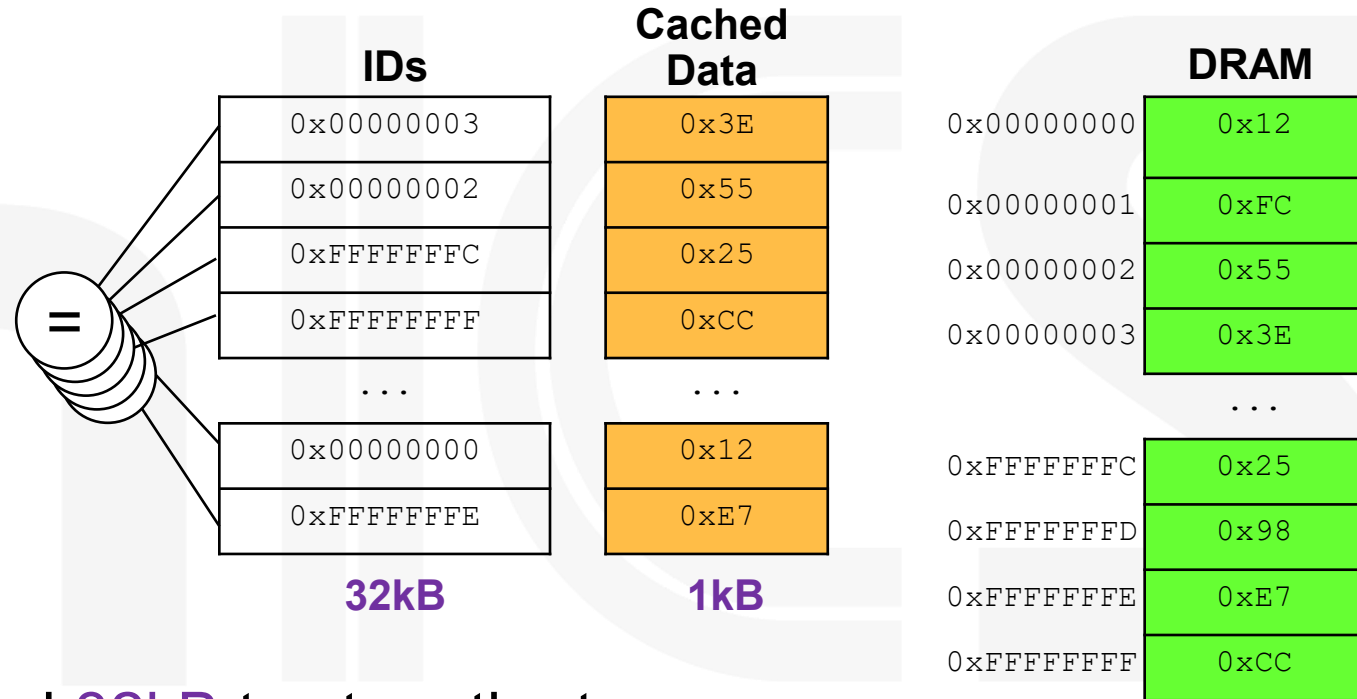


Goal: To reduce Average Memory Access Time (AMAT)

AMAT: $AMAT = HitTime + MR \times MissPenalty$

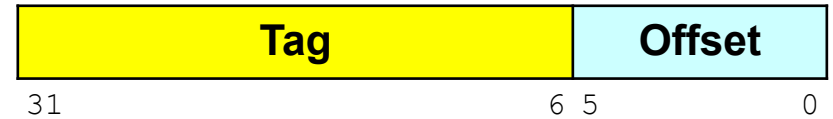
What are the IDs?

- We need to provide an ID to tag each unit of data
 - Why not just use the address?
- So, in an RV32 machine:
 - Address is 32-bits
 - Byte addressable
 - We need 32-bits to tag a byte
- Example:
 - Let's cache 1kB (2^{10}) of data.
 - We need 1kB to store the data and 32kB to store the tags.
 - We also need to compare our 32-bit address with all 1024 tags.
- Seems a bit costly, doesn't it?



Caching blocks of data

Address



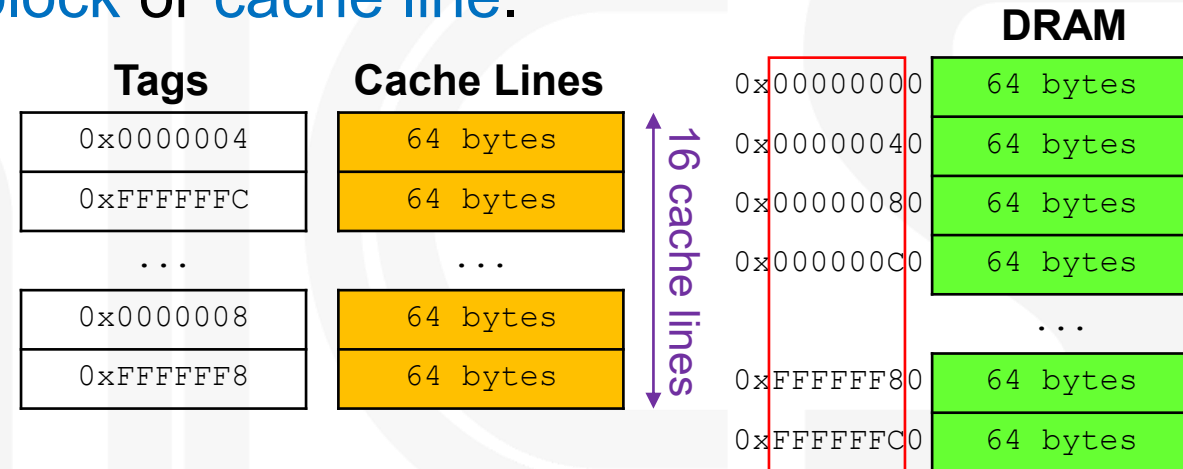
- Instead, why don't we store **chunks of data** together
 - For example, we can bring in **several bytes** starting at a **given address**.
 - We call such a chunk of data a **cache block** or **cache line**.

- **Let's revisit our example:**

- We want to cache **1kB** (2^{10}) of data.
- We will use **64B** (2^6) cache lines.
- We have a total of $2^4=16$ cache lines.

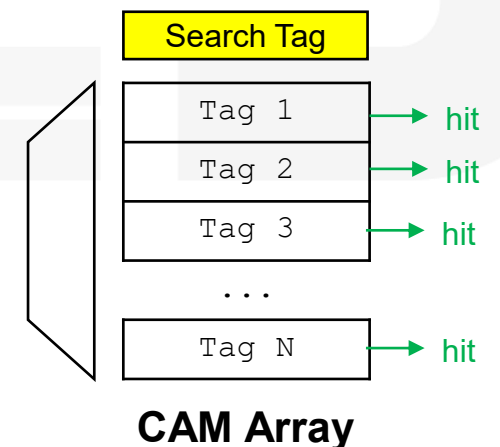
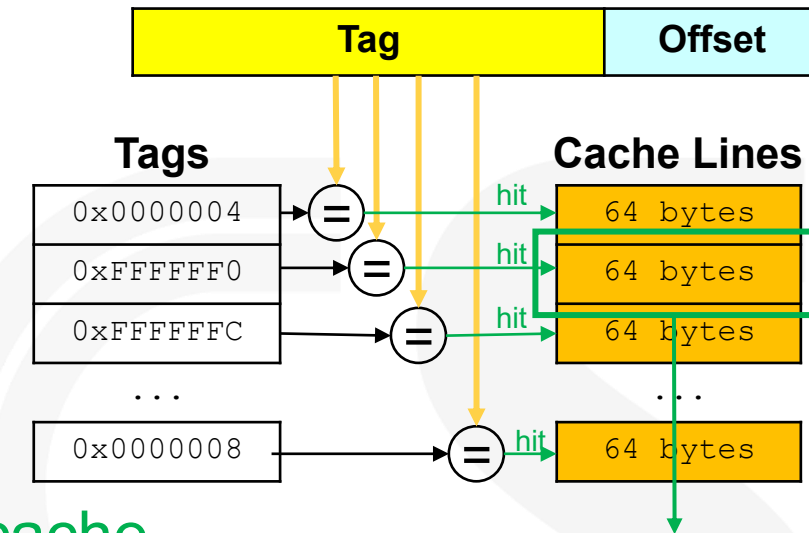
- **What about the IDs?**

- If we keep our cache lines aligned, the first **26-bits** in the addresses of all bytes in a cache line **are the same**.
- So, we'll use only **26-bits** to **identify the cache line**. This is called a **tag**.
- The remaining **6-bits** are used to **find the byte in the line**. This is the **offset**.



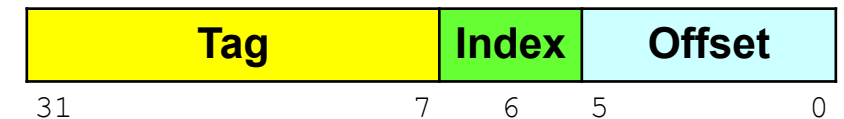
Fully Associative Cache Lookup

- When accessing memory, we have to check if the requested address is in the cache.
 - We need to compare the tag of the address with **all of the tags** stored in the cache.
 - This is known as “**fully associative**” cache lookup, since **any cache line** can be stored **anywhere in the cache**.
- Fully associative caches are **very expensive**
 - Basic implementation: **Comparator** for each cache line.
 - Circuit optimization: **Content-Addressable Memory (CAM)**
- Can we reduce the hardware cost?



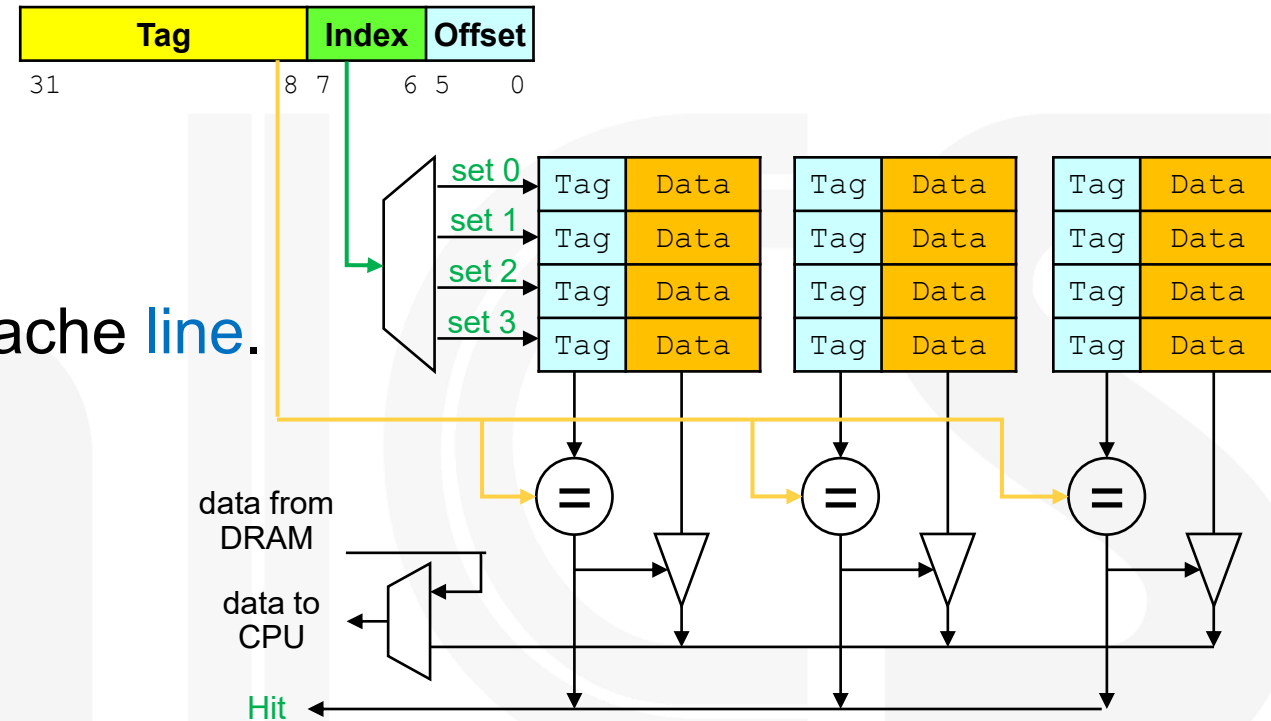
Set-Associative Caches

- What if we **reduce** the level of associativity?
 - For example, divide the cache into two “**sets**”
 - A certain cache line can be stored in **only one of the two sets**.
 - Now we need **only half** the number of comparators.
- Let's take our previous example:
 - 1kB of cache, 64B cache lines
 - Divide the 16 lines into **two sets of 8**. We need 8 comparators instead of 16.
 - The tag can be in **8 places in the set**. We call this “**8-way set associative**”
- How do we know which set our cache line is in?
 - Use **bit 6** of the address to select the set.
 - This bit is called the **index** bit.
 - If we have more sets (e.g., 4, 8, 16), use more **index bits**.



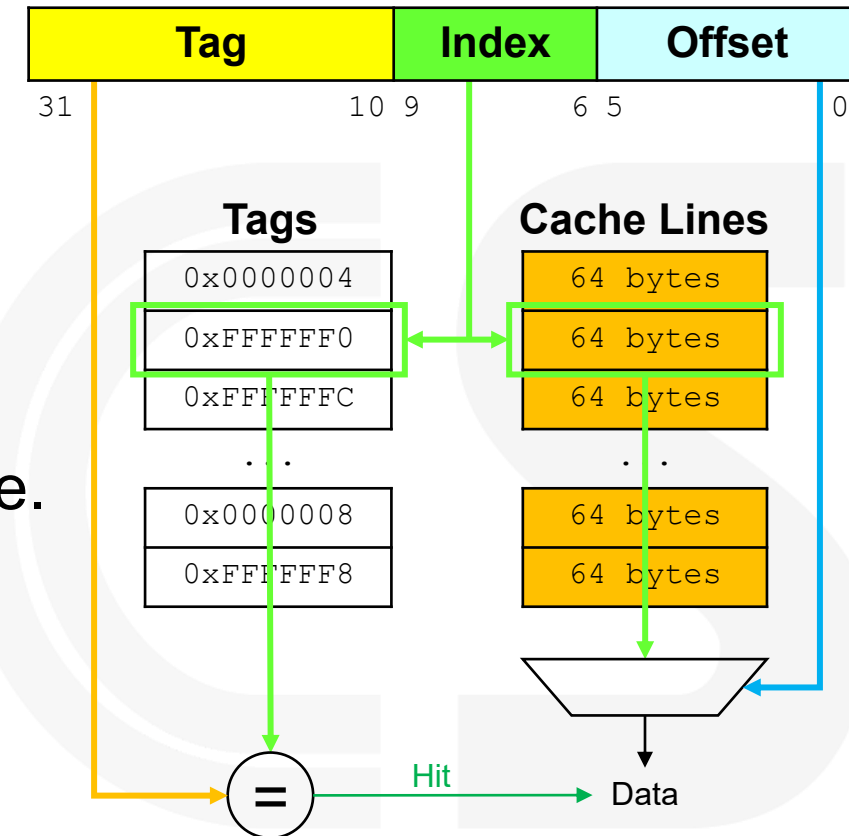
Set-Associative Cache Lookup

- Given a cache with L lines
 - Divide it into S sets ($\log_2 S$ index bits)
 - Each set includes L/S lines
 - So, there are $N=L/S$ ways for each cache line.
- The index bits are used to select a set
 - Drive the index bits into a decoder.
 - #Sets = #Rows
 - Accordingly, #Ways = #Columns
 - Each column has a comparator.
 - Select a set and compare address with all tags in the set.
- This example: a 3-Way Set Associative Cache
 - $L=12$ cache blocks, $S=4$ sets, $N=3$ ways.



Direct-Mapped Caches

- What is the maximum number of sets we can have?
 - That is when a set has **only one cache line** in it.
 - In this case, a given cache line can be stored in only one place, i.e., “**1-way set associative**”. Only one comparator is needed.
 - This type of cache is called a “**direct-mapped**” cache.
- Back to our example:
 - 1kB cache, 64B lines, 16 cache lines.
 - A **direct mapped cache** will have 16 sets → 4 index bits.
 - The **index bits** are used to read out one tag and one cache line.
 - The **tag** is compared to the address to identify **hit** or **miss**.
 - The **offset** is used to choose the byte/word from the cache line.



Summary: Alternatives in an 8 Block Cache

- Given a cache with 8 blocks, what are the associativity options?

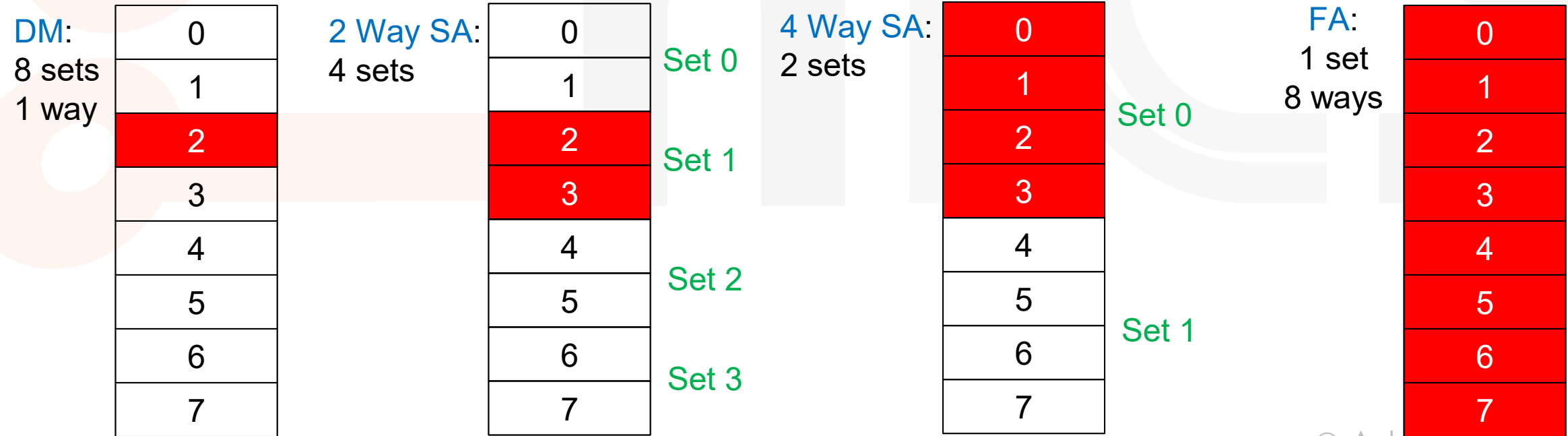
- Direct Mapped: 8 blocks, 1 way, 1 tag comparator, 8 sets
- 2-Way Set Associative: 8 blocks, 2 ways, 2 comparators, 4 sets
- 4-Way Set Associative: 8 blocks, 4 ways, 4 comparators, 2 sets
- Fully Associative: 8 blocks, 8 ways, 8 comparators, 1 set

Tag	3 Index bits	Offset
-----	--------------	--------

Tag	2 Index bits	Offset
-----	--------------	--------

Tag	1 Index bit	Offset
-----	-------------	--------

Tag	Offset
-----	--------



Summary: Cache Addressing Terminology

- To summarize, we divide our address into three parts:

- **Offset**

- Specifies which **byte** within the **block (line)** we want

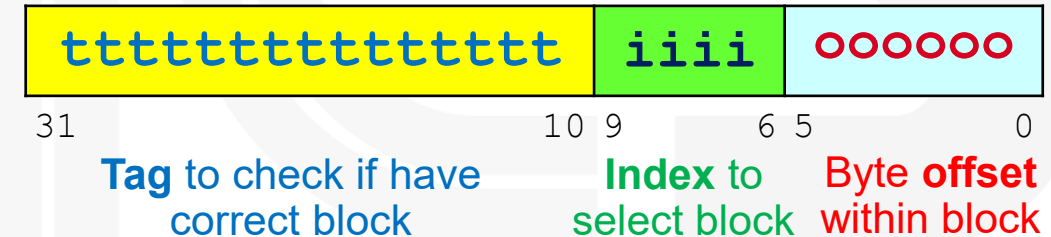
- **Set Index**

- Select which set to search in.
 - **Size of Index** = $\log_2(\text{number of sets})$

- **Tag**

- The remaining bits.
 - Used to distinguish between all the memory addresses that **map to the same location**
 - **Size of Tag** = Address size – **Size of Index**
– $\log_2(\text{number of bytes/block})$

Address





Tradeoffs in Cache Design



The “3-C’s” of Caching

- Remember, the goal of caching is to minimize **AMAT**

$$AMAT = HitTime + MissRate \times MissPenalty$$

- To achieve this, we will need to make tradeoffs in **cache organizations**.
- The primary factor affected by different cache organizations is **Miss Rate**.
- To analyze miss rate, we will define **three types of cache misses**:
 - **Compulsory Misses**: These are cold start or first reference misses.
The first (number of) times you access the cache, you will always get a miss.
 - **Capacity Misses**: These are misses due to cache size.
A cache often cannot contain all the data that the program needs.
 - **Conflict Misses**: These are misses due to the mapping of several addresses to the same cache set.

The “3-C’s” of Caching

• Compulsory Misses

- Compulsory misses always occur **after reset** or a **cache invalidation**.
- Compulsory misses **cannot be avoided**.
- However, if running billions of instructions, compulsory misses **are insignificant**.
- Increasing block size will **reduce compulsory misses**.

How to simulate?

Set cache size to **infinity** and **fully associative**, and count number of misses

• Capacity Misses

- Capacity misses are directly related to **cache size**.
- Capacity misses **would not occur** with an **infinite cache**.
- Larger caches usually **increase access time**.

How to simulate?

Change cache size from infinity and count misses for each reduction in size

The “3-C’s” of Caching

• Conflict Misses

- Fully-associative caches allow a memory address to be stored within any cache line. Therefore, there are no conflict misses.
- However, N-Way set associative caches only allow an address to be stored within a specific set. If the set is full, a conflict miss occurs.
- The worst example is direct-mapped caches, where each set has only one way.

• Example:

- 1kB direct-mapped cache with 64B blocks
- Program increments a variable at address 0xF500
- Instruction loop at addresses 0x0100-0x010C
- Every memory access misses in cache!

How to simulate?

Change from FA to n-way set associative while counting misses

Word Address	In Binary (index)	Hit/Miss
0x100	0000 00 01 0000 0000	Miss
0xF500	1111 01 01 0000 0000	Miss
0x104	0000 00 01 0000 0100	Miss
0xF500	1111 01 01 0000 0000	Miss
0x108	0000 00 01 0000 1000	Miss
0xF500	1111 01 01 0000 0000	Miss
0x10C	0000 00 01 0001 0000	Miss
0xF500	1111 01 01 0000 0000	Miss
0x100	0000 00 01 0000 0000	Miss

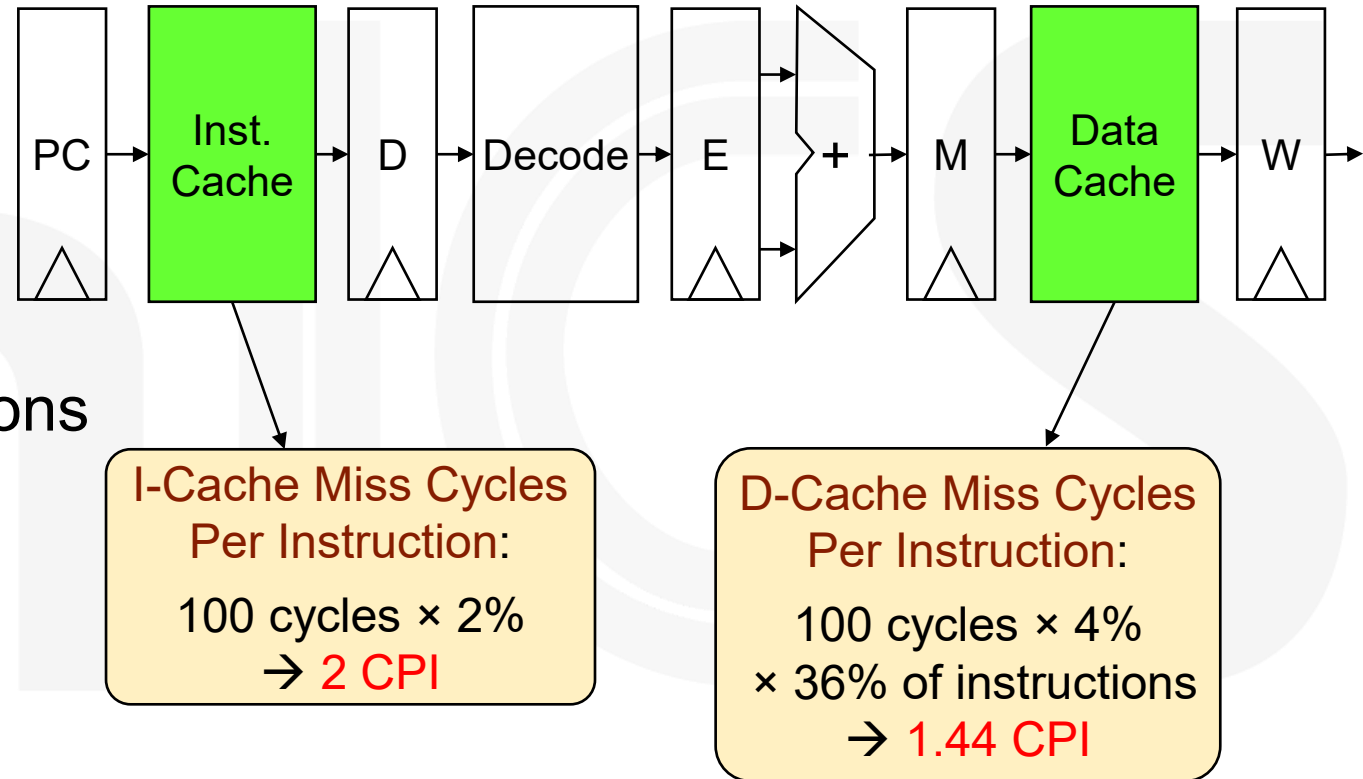
Cache Performance Example

- **Given**

- Base CPI (ideal cache) = 2
- Instruction Cache miss rate = 2%
- Miss penalty = 100 cycles
- Data Cache miss rate = 4%
- Load & stores are 36% of instructions

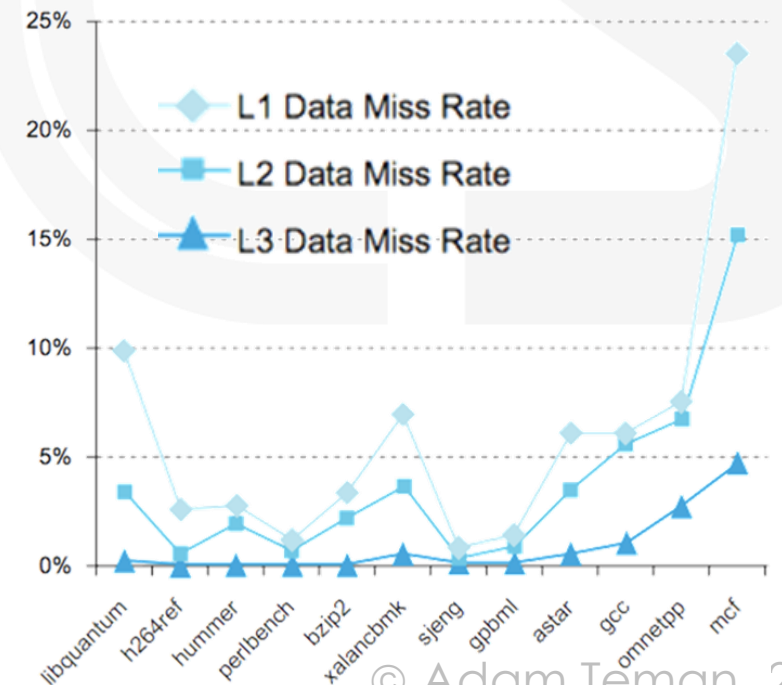
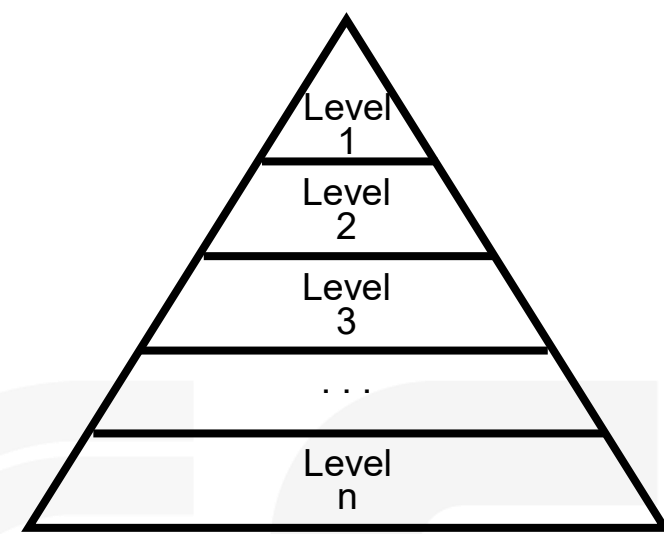
- **Actual CPI: $2 + 2 + 1.44 = 5.44$**

- Ideal CPU is $5.44/2 = 2.72$ times faster



Multilevel Caches

- To improve cache performance, use a **hierarchy** of caches
- **Local Miss Rate**
 - Fraction of misses at a given level of a cache
 - Local Miss rate $L2\$ = L2\$ \text{ Misses} / L1\$ \text{ Misses} = L2\$ \text{ Misses} / \text{total_L2_accesses}$
- **Global Miss Rate**
 - Fraction of misses that go all the way to memory
 - Global Miss Rate $= L_N \text{ Misses} / \text{Total Accesses}$
 - $L_N\$$ local miss rate \gg than the global miss rate
- **Design Considerations**
 - **L1\$**: **Fast access** \rightarrow min hit time \rightarrow small cache
 - **L2\$, L3\$**: **Low miss rate** (reduce DRAM access) \rightarrow large cache, block size, associativity



Multilevel Cache Example

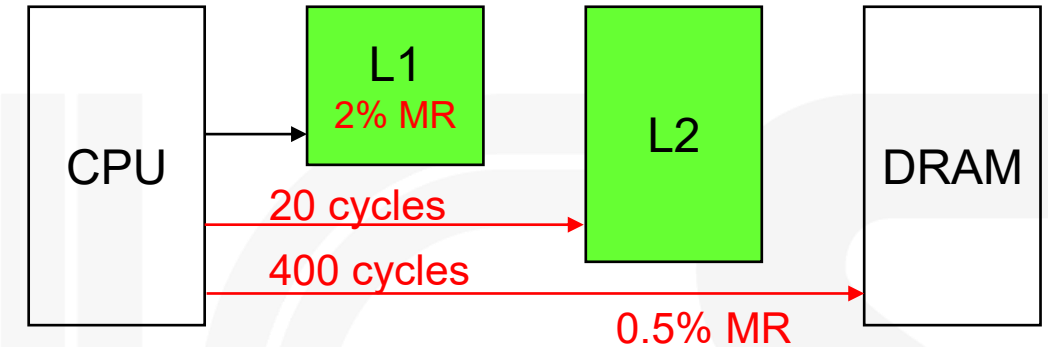
- **Given**

- Base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns
- Miss penalty = $100\text{ns}/0.25\text{ns} = 400$ cycles

- **Now add L2 cache**

- L2 Access time = 5ns
 - Miss penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles.
- Global miss rate = 0.5%

- **Performance improvement: $9/3.4 = 2.6\times$**



With one level of Cache:

$$\text{CPI} = 1 + 400 \text{ cycles} \times 2\% \\ \rightarrow 9 \text{ CPI}$$

Adding Level-2 Cache:

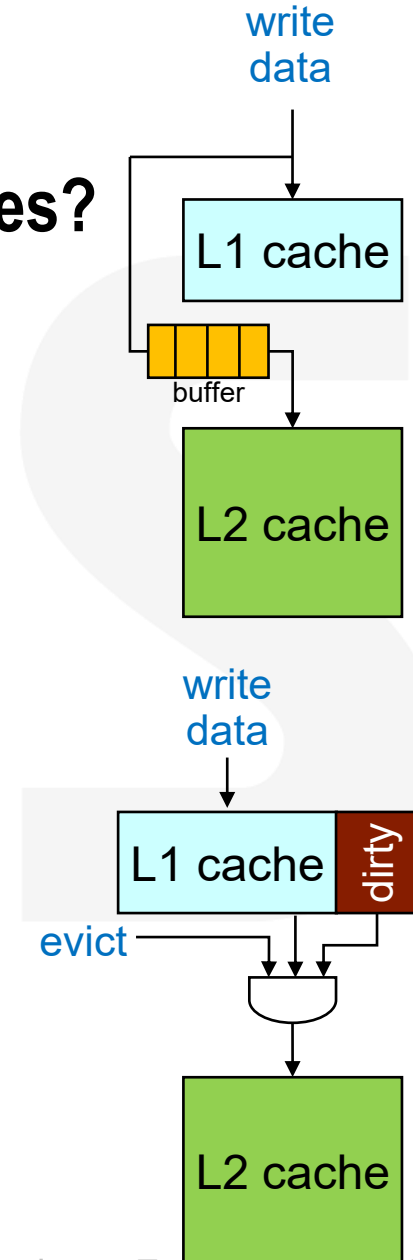
$$\text{CPI} = 1 + 20 \text{ cycles} \times 2\% + 400 \text{ cycles} \times 0.5\% \\ \rightarrow 3.4 \text{ CPI}$$

Cache Line Replacement Policy

- When we have a **capacity/conflict miss**, which block to we evict?
 - For **direct-mapped cache**, there is **no choice** to be made.
 - For **N-way associativity**, a **replacement policy** must be implemented.
 - Tradeoff **miss-rate reduction** vs. **complexity of implementation**.
- **Common replacement policies:**
 - **Random Replacement**: Randomly select a cache line to evict
 - Not bad performance. **Simple to implement**.
 - **Least-Recently Used (LRU)**: Replace the temporally least accessed entry.
 - **Great performance**. **Hard to keep track** of access order.
 - **For 2-ways**, a **single bit** is needed to implement true LRU.
 - **Pseudo-LRU**: Approximate LRU, e.g., “**not most recently used**”
 - Good performance. **Much easier** to implement than true LRU.
 - Many different implementations proposed

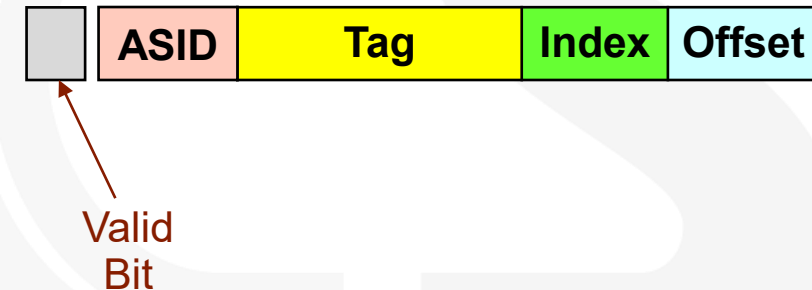
Write Policy

- How do we make sure cache and memory *have same values* on writes?
- **Write-Through Policy:**
 - Write cache and **write through** the cache to memory
 - Too slow, so include **Write Buffer** to allow processor to continue
 - Write buffer may have multiple entries to absorb bursts of writes
- **Write-Back Policy:**
 - Write **only to cache**. Write block back to memory **when evicted**.
 - Only **single write to memory** per block
 - Need to specify if block was changed → include “**Dirty Bit**”
- What do you do on a **write miss**?
 - Usually **Write Allocate**
→ First fetch the block, then write and set dirty bit.



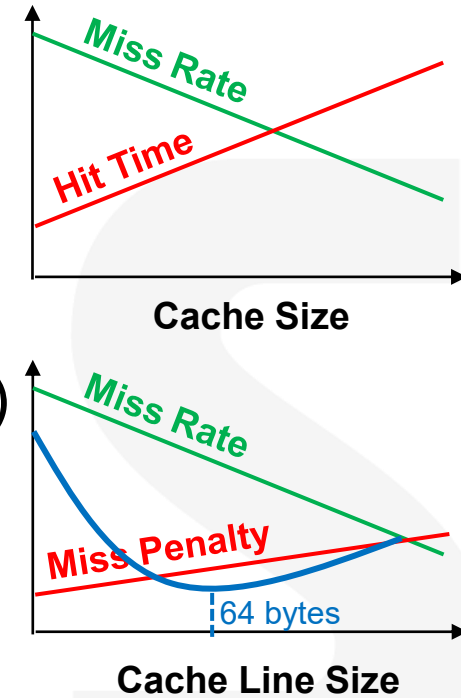
One More Detail: Valid Bit

- When a new program starts, cache data is garbage.
 - Need an indicator whether this tag entry is valid for this program
- Add a “**valid bit**” to the cache tag entry
 - 0 => cache **miss**, even if by chance, address = **tag**
 - 1 => cache **hit**, if processor address = **tag**
- **Cache invalidation**, means that all valid bits are reset.
 - Cache invalidation is done upon **reset**.
 - But it is also done for **cache coherency**, when a different process writes to their local copy of the same physical address.
- An additional concept is a “**Cache Flush**”
 - This incurs writing back **dirty data** and (sometimes) **invalidating** the cache.
 - Caches may be flushed upon **context switch**, but **ASID** helps avoid this.



Summary: Cache Design Trade-Offs

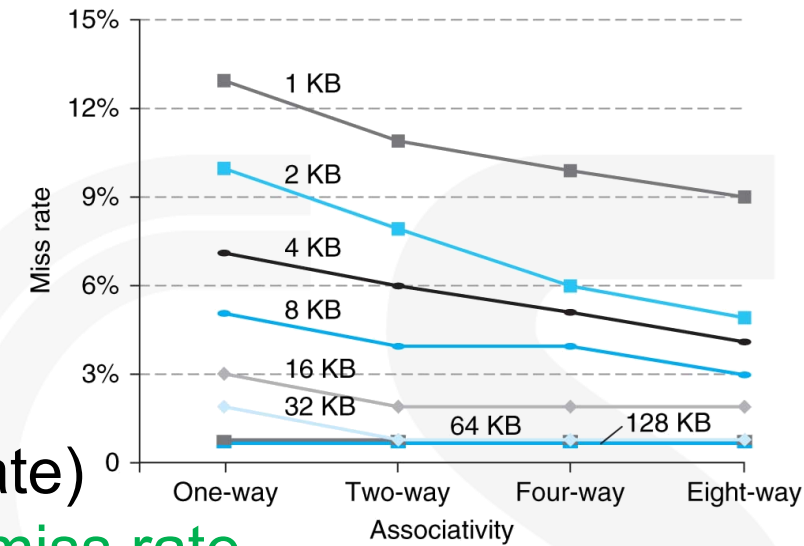
- We've now seen many design choices in cache design.
 - Let's think about how these choices affect our cache.
- **Cache Size**
 - The larger the cache, the **fewer capacity misses** (lower miss rate)
 - However, larger caches lead to **increased hit time**.
- **Cache Block Size**
 - Large cache lines lead to **increased spatial locality** (reduced miss rate) as well as **shorter tags** and **fewer compulsory misses** (negligible).
 - But there are fewer blocks in the cache, leading to **reduced temporal locality** (increased miss rate), and bringing in larger blocks **increases the miss penalty**.
 - A good sweet spot has been found to be **64B** – now commonly used.



Summary: Cache Design Trade-Offs

• Associativity

- Fully-Associative caches **eliminate conflict misses** (reduced miss rate), but they are **expensive to implement** (comparators/CAMs)
- Direct-Mapped caches are **much cheaper** but suffer from **frequent conflict misses** (increased miss rate)
- Going from DM to 2-way provides **20%+ reduction in miss rate**.
- **Little miss-rate benefit** going beyond 4-8 ways and **hit time increases**.



• Write Policy

- Write Through: **simple**, **predictable**, **reliable**, but **many writes to memory**
- Write Back: **lower bandwidth**, but **complex**, **less predictable**, **less reliable**

• Replacement Policies

- More “intelligent” (e.g., LRU) → **Lower miss rate**, but **complex implementation**

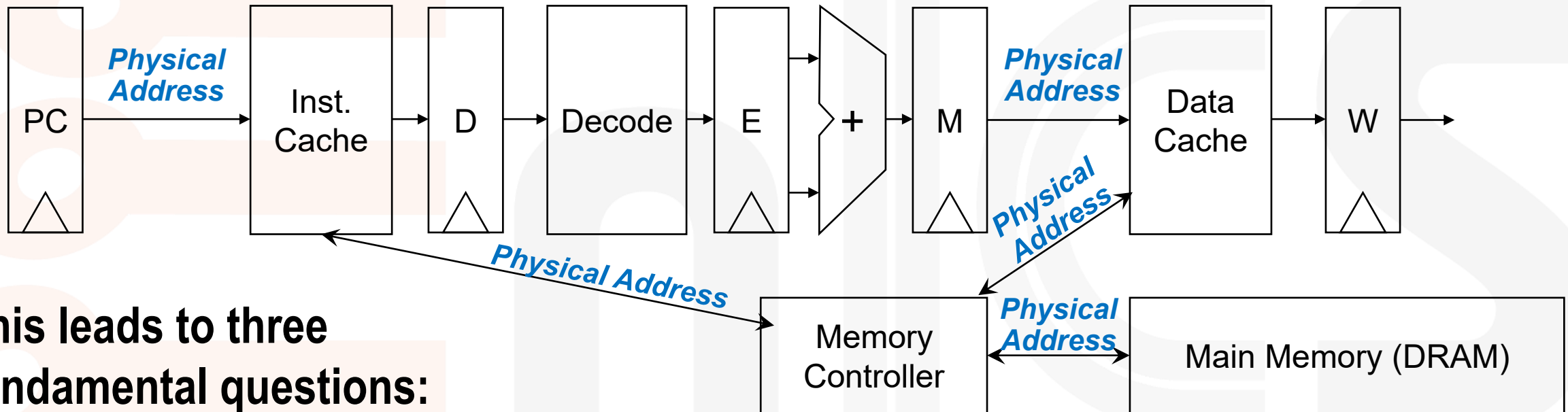


Virtual Memory



“Bare” 5-Stage Pipeline

- In a bare machine, the only kind of address is a **physical address**

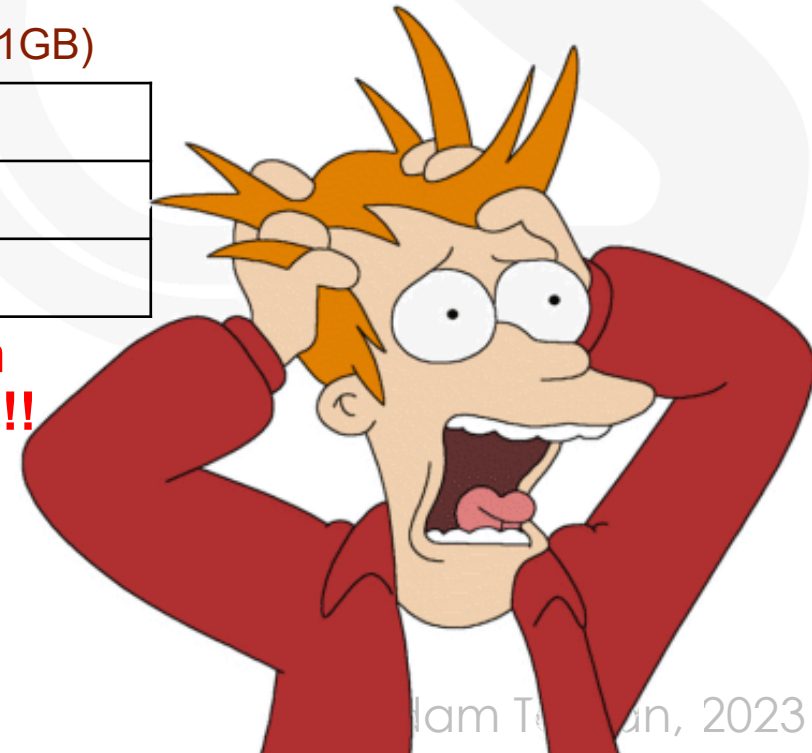
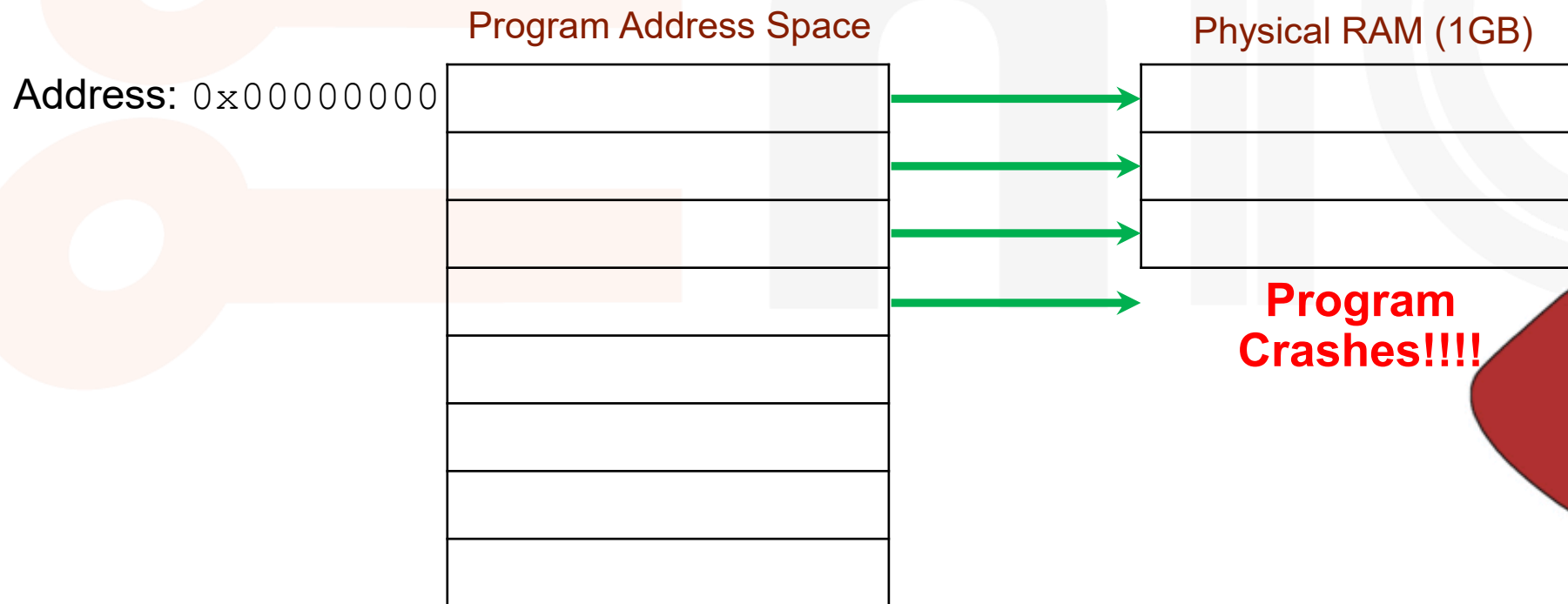
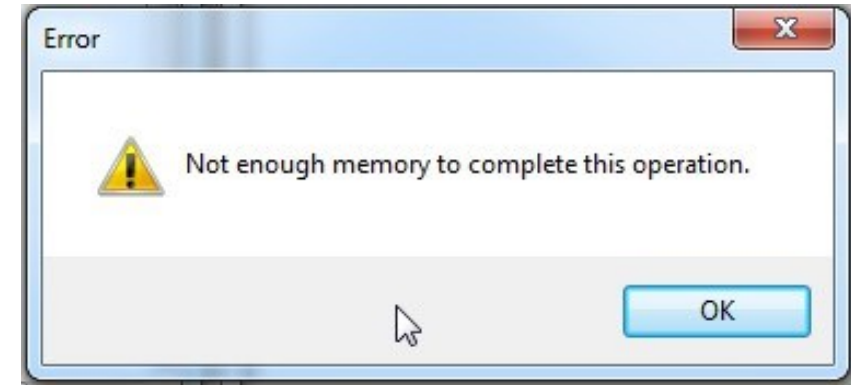


- This leads to three fundamental questions:

- What if we **don't have enough** memory?
- How do we allocate memory between **multiple processes**?
- How do we **isolate processes** from each other (and/or **share data**)?

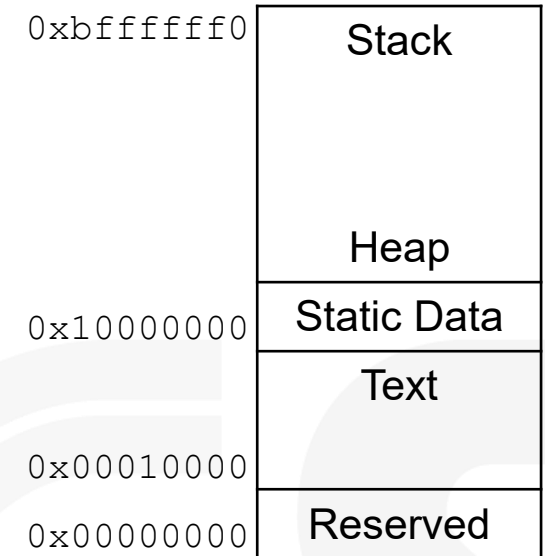
Problem #1: Memory Size

- RV32 has a **32-bit** address space $\rightarrow 2^{32}=4\text{GB}$
 - A program can access *any* of these 2^{32} bytes.
 - What if you don't have **4GB** of memory?
 - What if you want to run multiple programs simultaneously?



Problem #2: Process Isolation

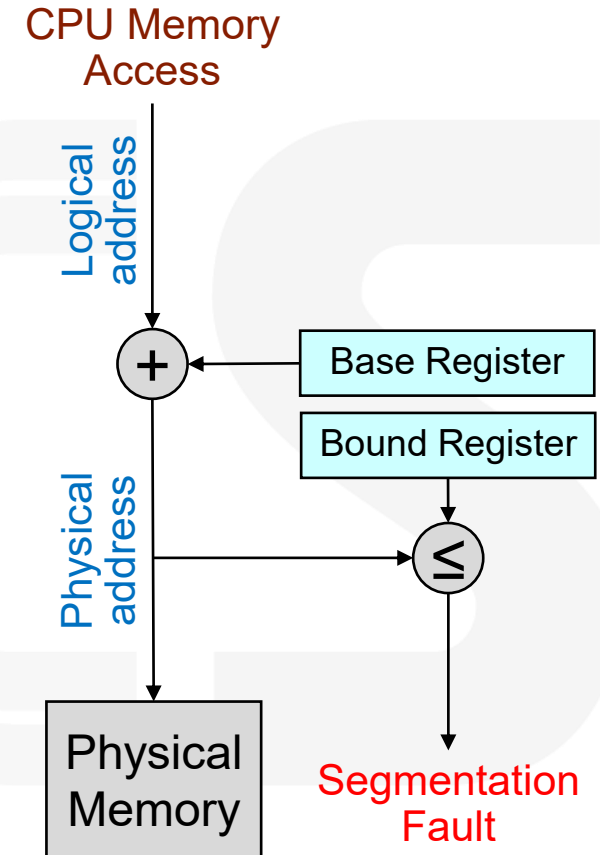
- When we discussed the build process, we introduced the concept of a **Memory Map**.
 - A program has a 2^{32} byte address space.
 - Different parts of the address space were reserved for different usages (e.g., program code, stack, global variables, I/O...)
- But what if we want to run multiple programs simultaneously?
 - Each “process” runs a program that *was built according to the memory map*.
 - Data from different processes *maps to the same address*.
 - Processes will **overwrite** data at the **same addresses**.
 - How can we **protect data** from one process being *read or written by another process*?



Name	CPU	Memory
> Google Chrome (27)	0.1%	2,631.5 MB
> Microsoft PowerPoint (2)	0%	292.4 MB
> Notepad++ : a free (GPL) sourc...	0%	1.4 MB
> PDF-XChange Editor	0%	101.9 MB
> Slack (8)	0%	289.5 MB
> Task Manager	0.1%	29.5 MB
> Windows Explorer	0%	137.3 MB
> Zoom Meetings (2)	0%	37.7 MB
Background processes (119)		
> Acrobat Collaboration Synchron...	0%	1.4 MB
> Acrobat Collaboration Synchron...	0%	3.5 MB

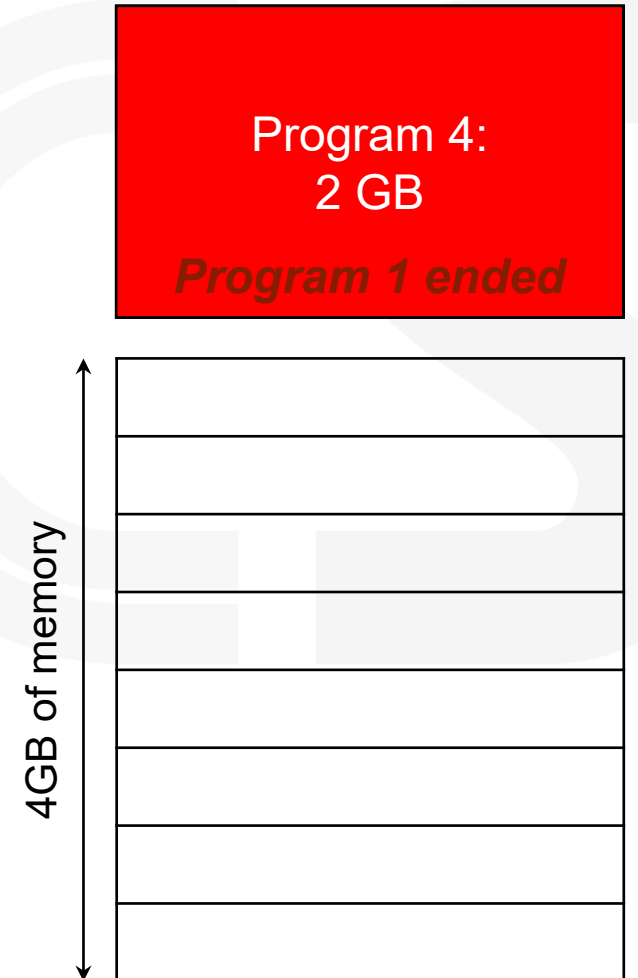
Possible Solution: Base and Bounds

- To address the **size problem**, we can **limit the address space** during compilation.
 - We will allocate fewer than 2^{32} bytes to each program.
- Then, when loading the program:
 - We will define **base** and **bound** addresses, where the program's address space will start and end.
 - When running a program, we will load these into a **base register** and a **bound register**.
 - When accessing memory, the **base register** will be added to the **logical address** to create the **physical address**.
 - The **physical address** will be compared to the **bound register**. If the bound is crossed, a **segmentation fault** exception will occur.



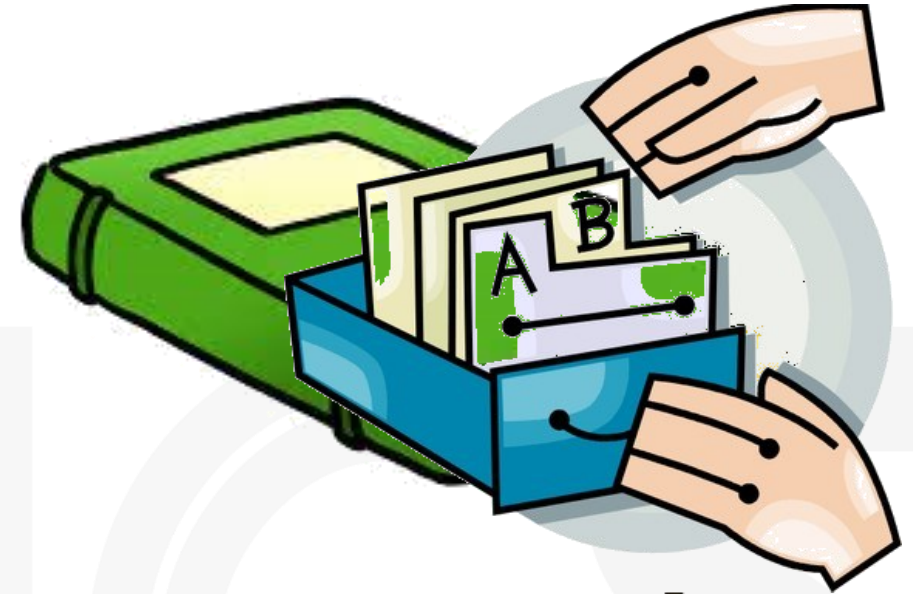
Segmentation

- The base and bounds approach is commonly known as “**Segmentation**”.
 - The **Operating System (OS)** allocates memory and sets the **base** and **bound** addresses.
 - Only the OS can update **base** and **bound** registers by running in a “**privileged mode**”
- **However, segmentation suffers from several problems:**
 - The address space of each process has to be **predefined** and **limited**.
 - Processes **cannot share** data.
 - Segmentation results in **fragmentation**.
- **Some architectures still use segmentation today**
 - For example, **x86** supports it, but it is rarely used.



Library book analogy

- An author writes a book and sells it to a library.
- The library puts the book on a shelf.
 - Does the author **print the location** of the book on the book cover?
 - **No**. Then we would need to set the location in **every library in the world**.
- Instead, we provide the book with an **identifier** (i.e., ISBN)
 - The library has a **catalog** that says where the specific book is placed.
- This **indirection** is the concept used to overcome our problems.

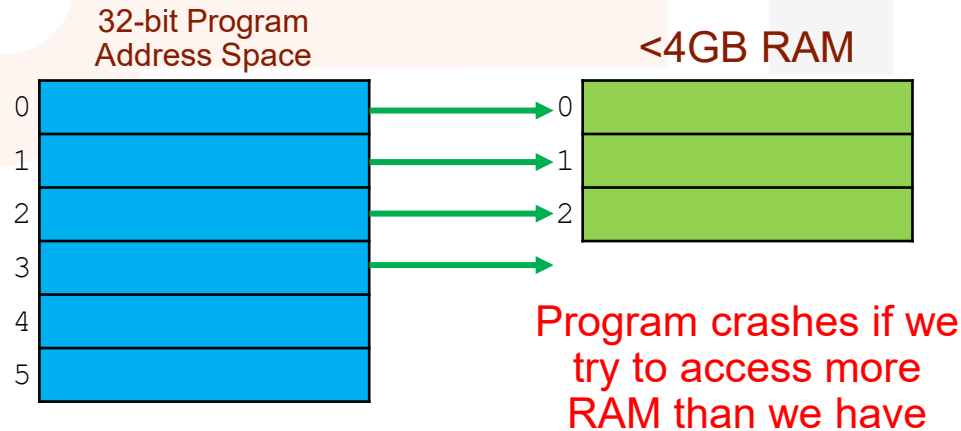


Solution: Virtual Addressing

- The fundamental theorem of software engineering (FTSE):
“*All problems in computer science can be solved by another level of indirection.*”
- Virtual addressing is an indirection that, indeed, solves many problems.
 - All programs own a virtual address (VA) space (of 2^{32} bytes)
 - The Operating System maps every VA to a physical address (PA).
 - The memory management unit (MMU) translates the VAs to PAs.

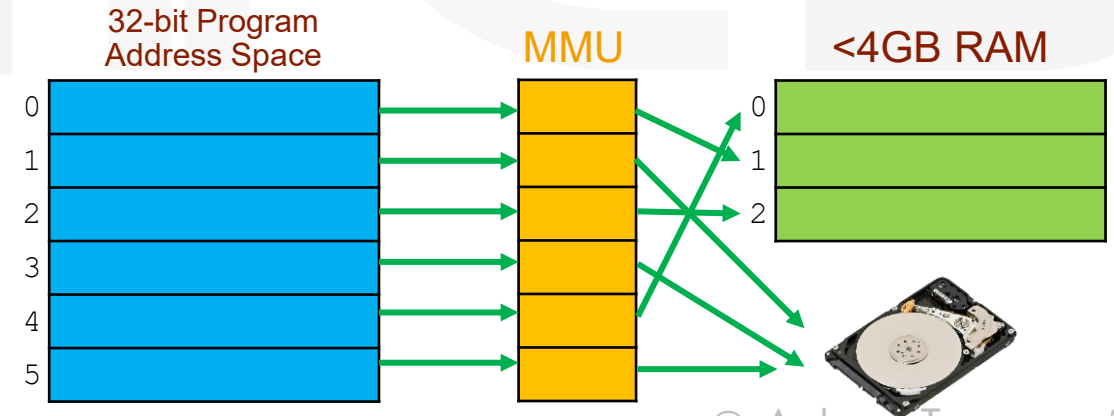
Without Virtual Memory

Program Address = RAM Address



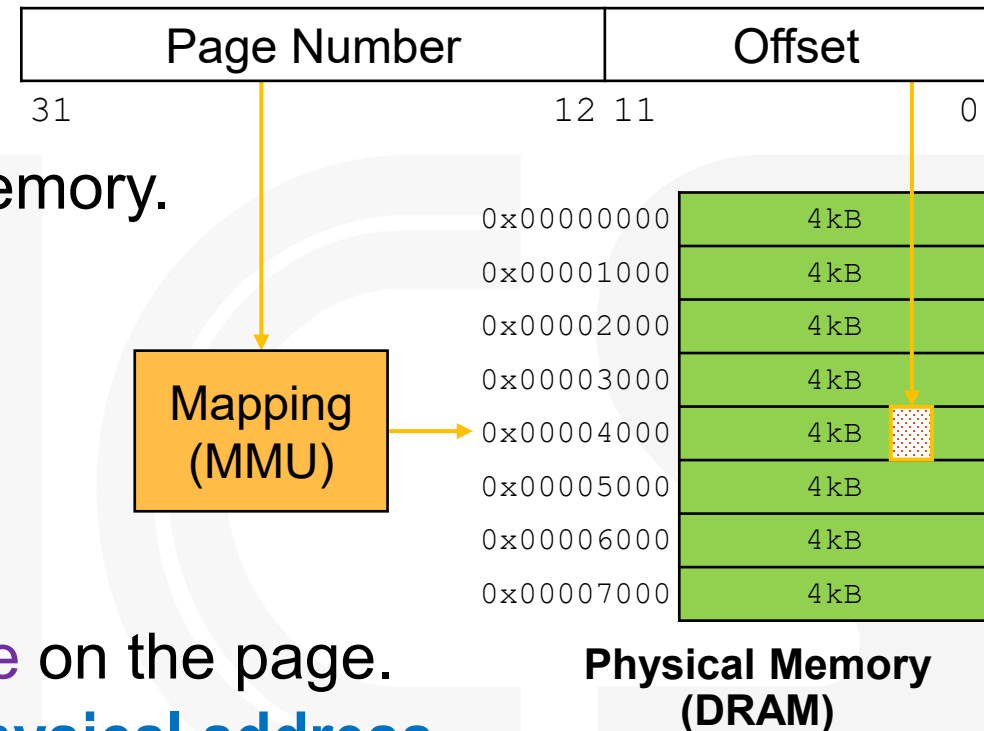
With Virtual Memory

Program Address Maps to RAM Address



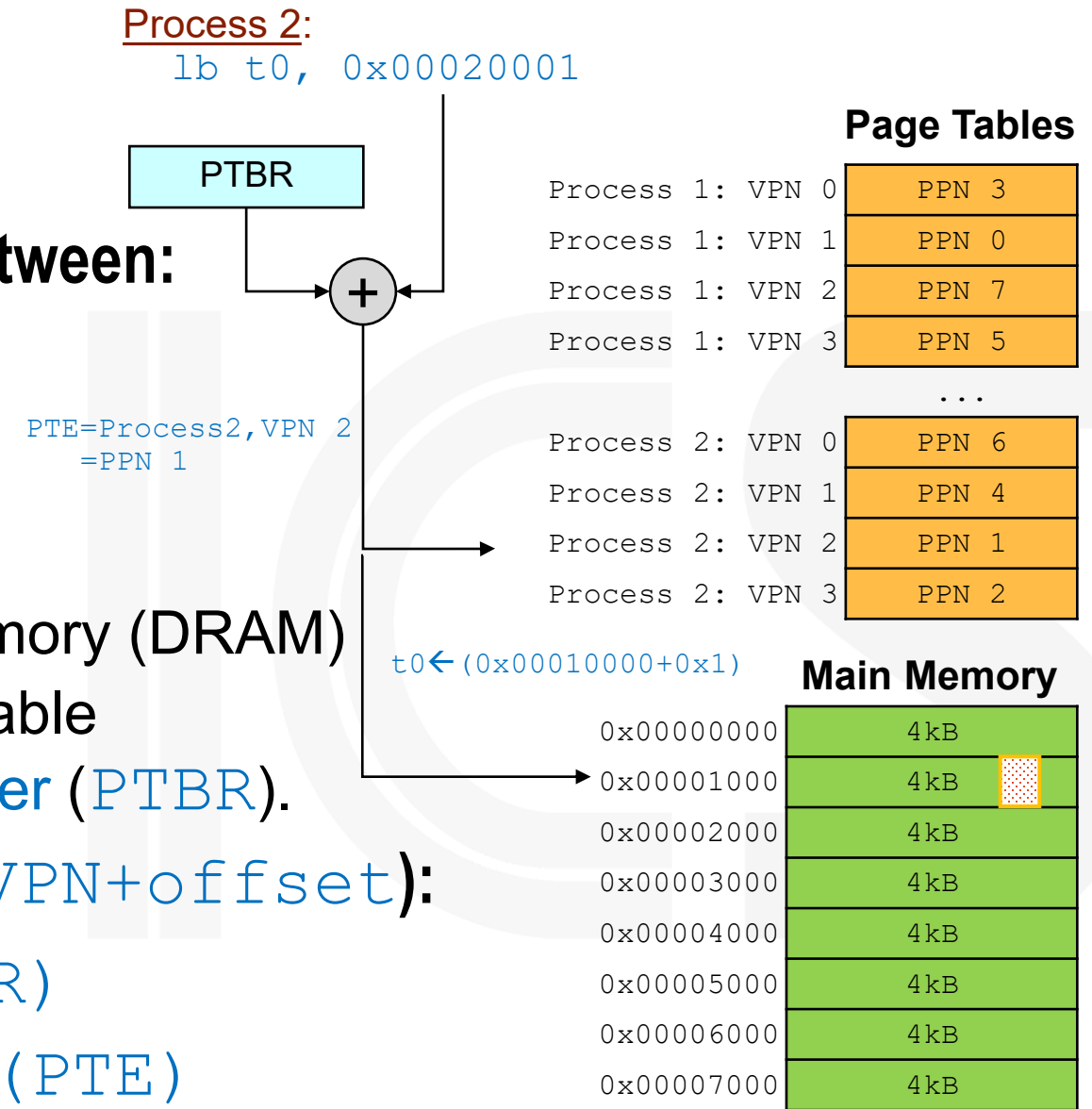
How does Virtual Addressing work?

- **Divide the memory into “pages”**
 - A page is a unit of reference in the physical memory.
 - Typical page size is 4kB (12 bits)
- **The processor accesses the memory with a “Virtual Address”**
 - The VA is divided into a “page number”, which is an identifier of the *virtual page* and an “offset”, which is the *number of the byte* on the page.
- **The MMU translates the virtual address into a physical address**
 - The **page number** is replaced with the *base address* where the *virtual page* is stored in *memory*.
 - The **offset** is used to select the *byte* out of the *physical page*.
- **The mapping between virtual and physical pages is called a “page table”**



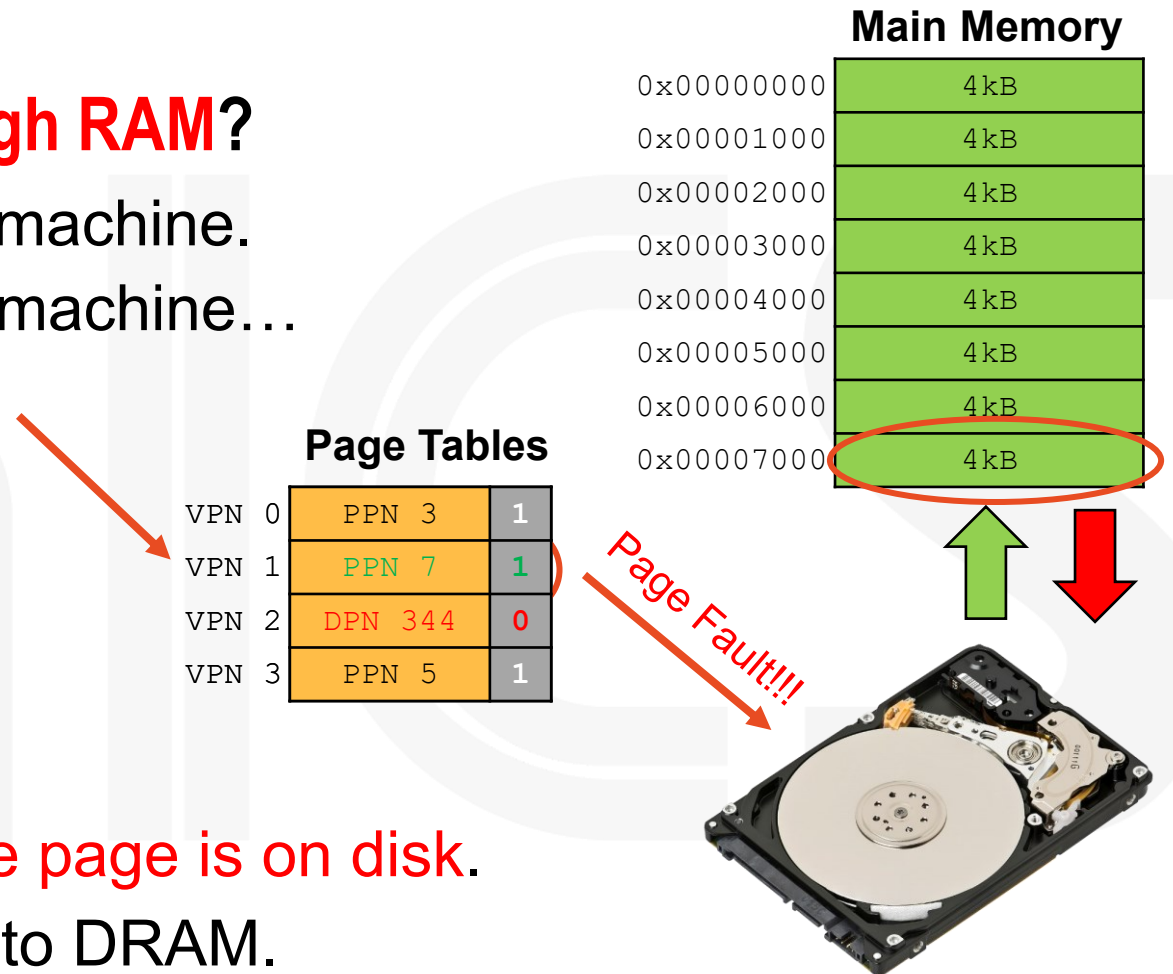
Page Tables

- A **page table entry (PTE)** is a mapping between:
 - Virtual page number (VPN)
 - and Physical page number (PPN)
- Each process receives its own page table
 - The page tables are stored in main memory (DRAM)
 - The base address of the current page table is stored in the **Page Table Base Register (PTBR)**.
- When accessing memory (e.g., `lb t0, VPN+offset`):
 - First DRAM access: $PTE \leftarrow VPN(PTBR)$
 - Second DRAM access: $t0 \leftarrow offset(PTE)$
- So (at least) **two DRAM accesses** required for every load/store operation!



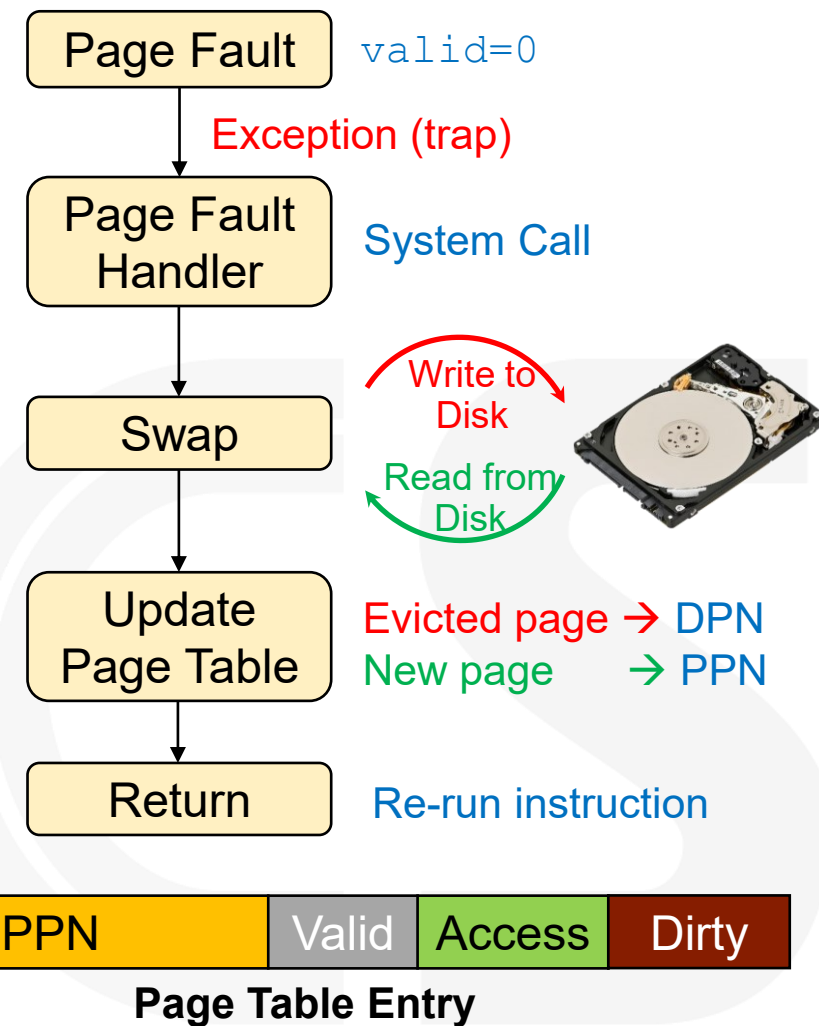
Demand Paging

- What happens when we **don't have enough RAM**?
 - For example, **1GB DRAM** on an **RV32** machine.
 - ...or running **100 processes** on such a machine...
- Let's just **use the DRAM as a cache**
 - Store "everything" on **disk**.
 - Only **bring accessed pages** to DRAM.
 - This is called "**demand paging**"
- Add a "**valid bit**" to our PTE
 - If a PTE with `valid=0` is accessed, **the page is on disk**.
 - A "**page fault**" occurs. Bring the page into DRAM.
 - **If DRAM is full**, *evict* a page, "**swapping**" it out to disk.
- Accessing disk takes **millions of cycles** → Use **Software (OS)**



Page Fault Handling

- What happens when you get a **page fault**?
 - This is a **hardware exception** → like an **interrupt**.
 - A **page fault handler** is called → the OS is invoked.
 - **If DRAM is full**, the OS chooses a page to **evict**.
 - **Swap page**: write back old page, read new page from disk to DRAM.
 - Update **page table**.
 - **Jump back** to rerun instruction that caused page fault.
- To **reduce cost** of page faults:
 - Use **fully associative** page placement (handled by OS)
 - Add “**access bit**” (a.k.a., “**use bit**”) to PTE to enable **pseudo-LRU**
 - Add “**dirty bit**” to PTE and only **write-back** swapped page when modified (= **dirty**)
 - **Never swap out** pages of the Operating System
- Or just **buy more memory!!!!**



Did Virtual Memory Solve our Problems?

- **Problem #1: Not enough memory**

- **Example #1: RV32 machine with 1GB DRAM.**

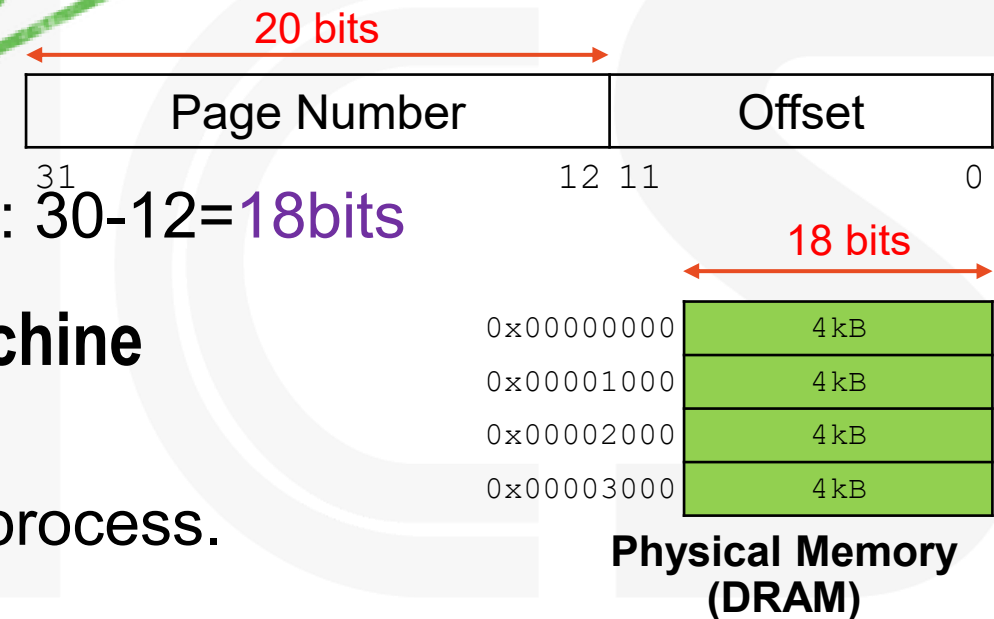
- Just use smaller **PPNs** than **VPNs**
- Assume **4kB** pages: **VPN**: $32 - 12 = 20\text{bits}$, **PPN**: $30 - 12 = 18\text{bits}$

- **Example #2: 100 processes** running on this machine

- Each process can use **10MB** of memory*.
- So we can allocate close to **2,500 pages** per process.

- **And what if that's not enough**

- Just go to disk
(at the cost of a lot of wasted cycles...)



* Not really, but a few MB, anyway

Did Virtual Memory Solve our Problems?

- **Problem #2: Program Isolation**

- **Example #1: Location Independence**

- Compile a program, assuming it has the entire 2^{32} byte address space.
- Each process has a **separate page table**, mapping the **same addresses to different physical locations**.

- **Example #2: Protection**

- Page table mapping ensures one process doesn't access the data of another process.

- **Example #3: Data Sharing**

- Map VPNs of several processes to the same PPN to enable data sharing.
- Add read (**R**), write (**W**), execute (**X**) bits to PPN for access restriction.

SOLVED

Page Tables

Process 1: VPN 0	PPN 3
Process 1: VPN 1	PPN 0
Process 1: VPN 2	PPN 7
Process 1: VPN 3	PPN 5
...	
Process 2: VPN 0	PPN 6
Process 2: VPN 1	PPN 4
Process 2: VPN 2	PPN 1
Process 2: VPN 3	PPN 2

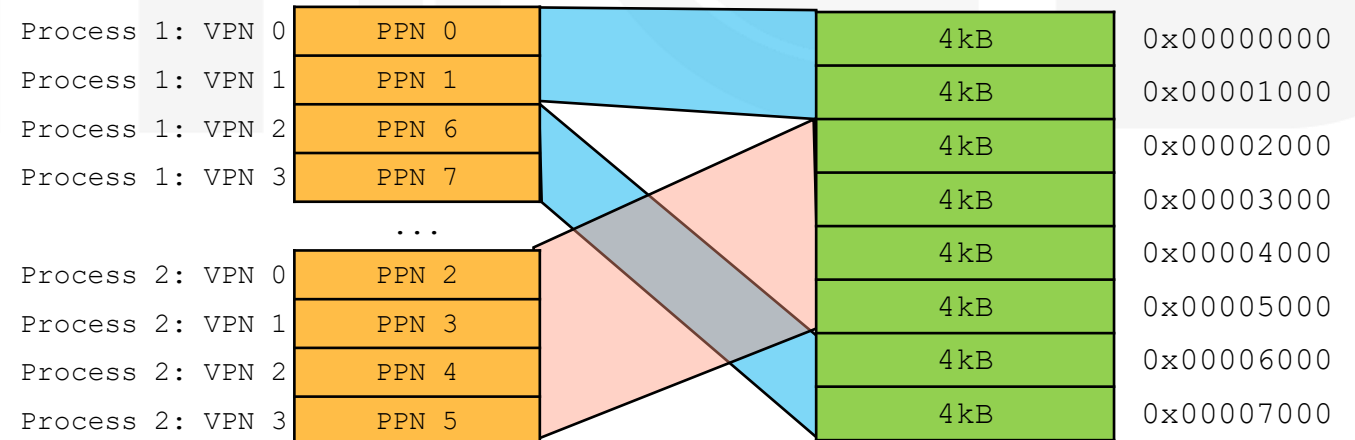
Did Virtual Memory Solve our Problems?

- **Problem #3: Fragmentation**

- **Example: Multiple Processes Running**

- Each process gets allocated 4kB pages upon access.
- The Operating System keeps account of free pages.
- Once a process exits, its allocated memory is freed.

- **Indirection completely solves fragmentation.**



SOLVED



Practical Paging

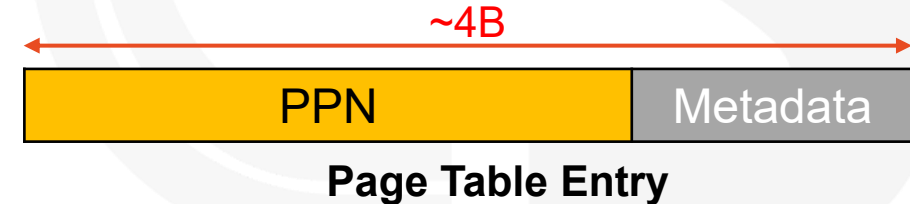


Are page tables even feasible?

- We know the following about **page tables**:
 - We need to provide a **VPN** for *every page* in the memory.
 - Each **page table entry** stores the **PPN** and **metadata**.
 - Page tables **cannot be swapped out** (otherwise, how can we find them?)
- With that in mind, **how much memory** do we need to store a page table?
 - Let's assume a **32-bit address** with **4kB pages**.
 - A **12-bit offset**, so the **PPN** size is (up to) **20-bits**.
 - Adding **metadata**, we need about **4B per PTE**.
 - Therefore, a **page table** requires about $2^{20} \times 4B = 4MB$
- Is that reasonable/feasible?
 - **Yes**. We probably have **GBs** of DRAM. We'll also need **4GB** of **swap**...
 - **But...** what if we have **100 processes**? **1000 processes**?

~4MB

VPN 0	PPN	R	W	X	...
VPN 1	PPN	R	W	X	...
VPN 2	PPN	R	W	X	...
VPN 3	PPN	R	W	X	...
...					
VPN $2^{20}-2$	PPN	R	W	X	...
VPN $2^{20}-1$	PPN	R	W	X	...



Possible Solution: Larger Pages



- Why do we keep assuming **4kB** pages?

- Well... **it's hard to get rid of old habits...**
- The **Intel 80386** (1985) supported **4kB** pages.

- But seriously... **small pages are good**

- The minimum OS allocation quanta is one page, so large pages can lead to **internal fragmentation**.
- Page faults are **cheaper** – less data transfer, less time.

- That being said, **larger pages are probably better...**

- More **spatial locality** → **Fewer page faults**
- Disks benefit from **burst operations**, so penalty for larger transfers small.
- ...and **smaller page tables**.

- But even **huge page tables** won't work for **64-bits...**

Small 4kB pages:

Page Number												Offset										
31												12	11									0

→ 4MB page table

Large 4MB pages:

Page Number																						Offset										
31																						22	21									0

→ 4kB page table

64-bit address, 4MB page, 8B PTE:

Page Number																						Offset										
63																						22	21									0

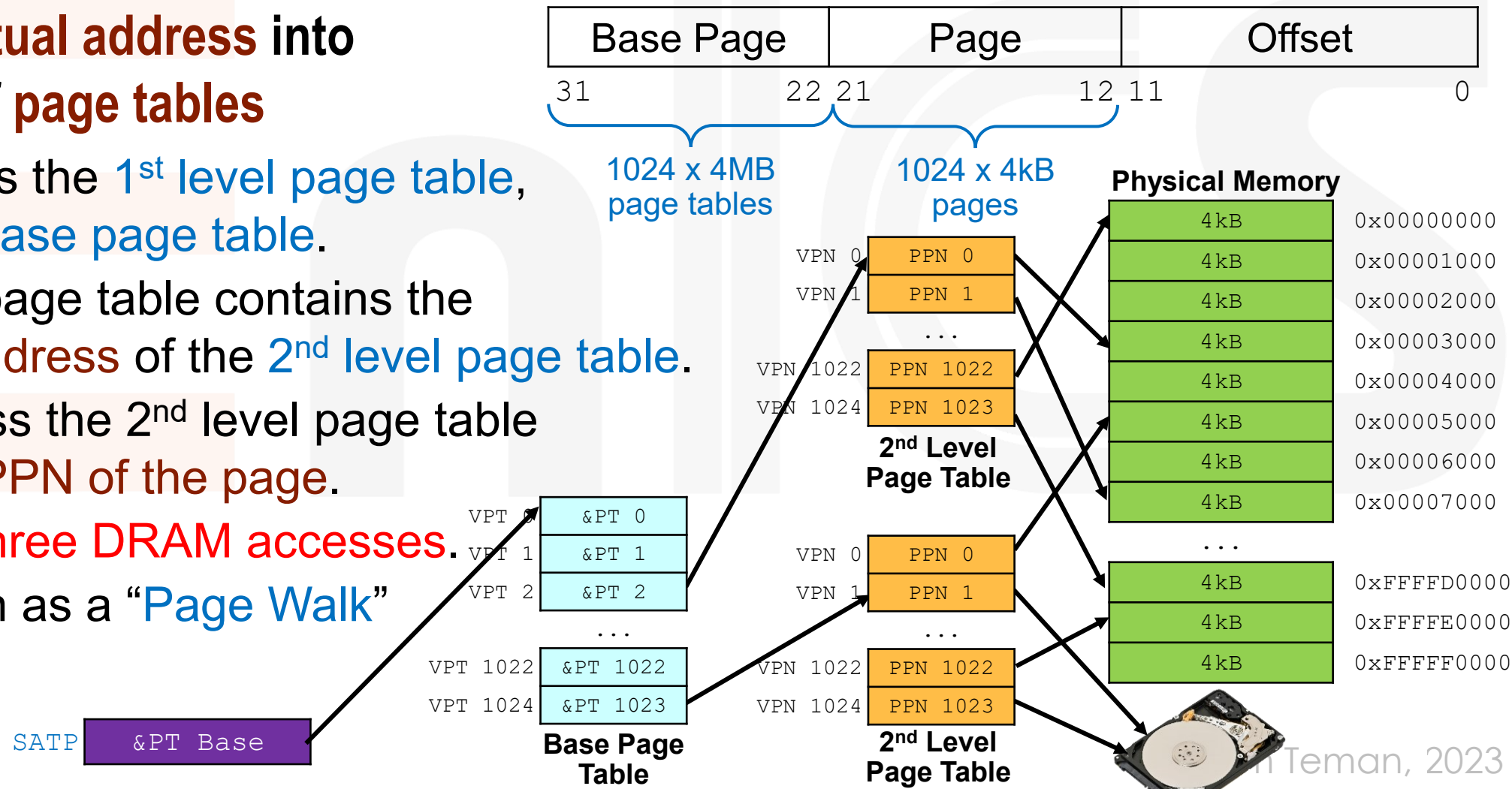
→ **32 TB** page table

Better Solution: Multi-Level Page Tables

- Of course, the solution to all problems is... **more indirection!**

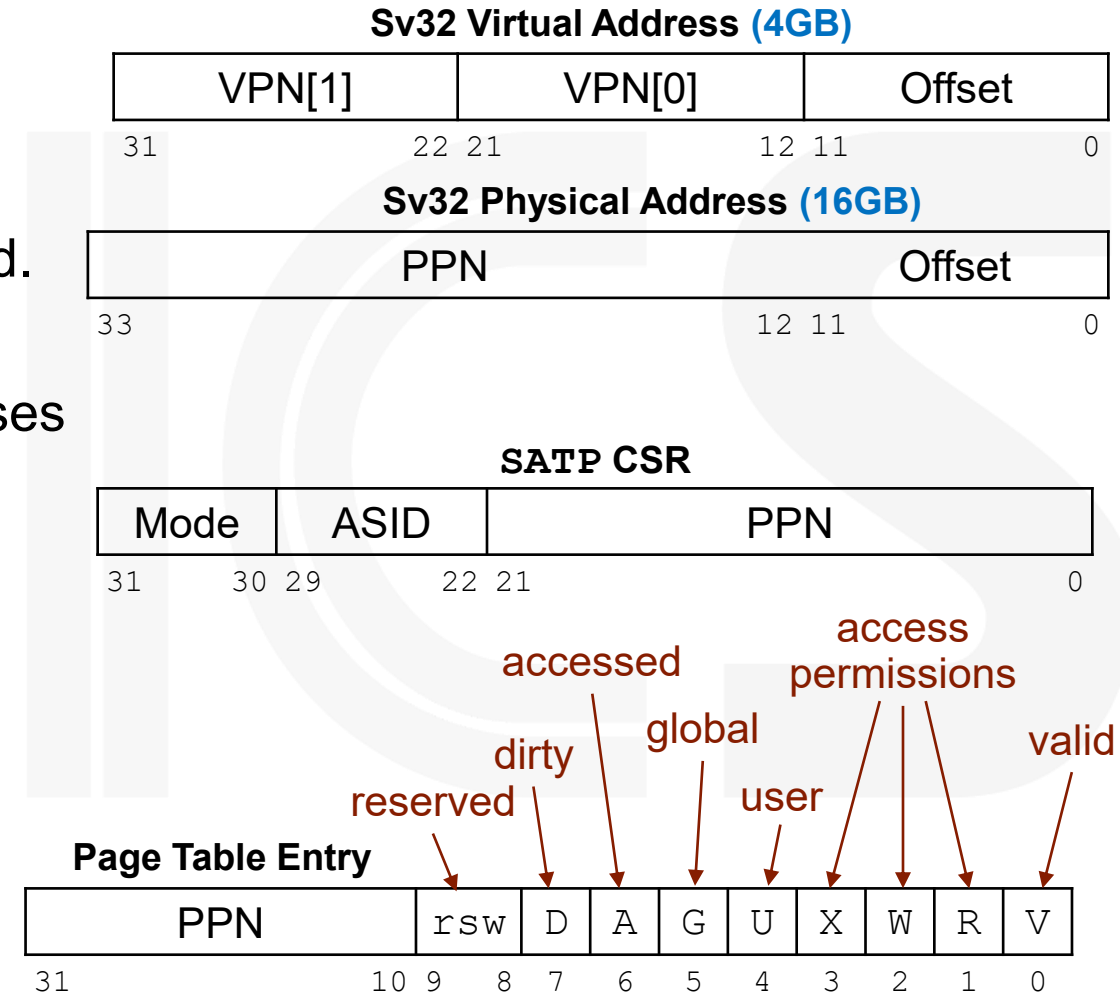
- Divide the **virtual address** into a **hierarchy of page tables**

- First access the **1st level page table**, a.k.a. the **base page table**.
- The base page table contains the **physical address** of the **2nd level page table**.
- Now, access the **2nd level page table** to get the **PPN** of the page.
- Requires **three DRAM accesses**.
- Also known as a “**Page Walk**”



Virtual Addressing in RISC-V

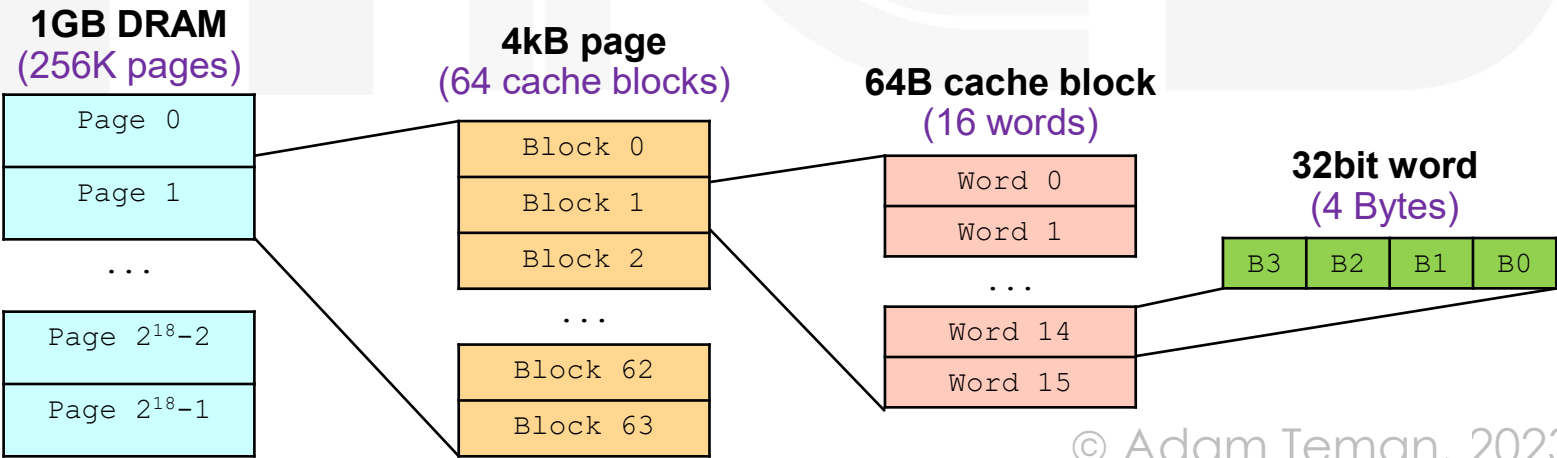
- RISC-V defines various specifications
 - For RV32: Sv32
 - Two-levels, 32bit Virtual Add., 34bit Physical Add.
 - For RV64: Sv39, Sv48, Sv57 ...
 - More levels, longer Virtual and Physical Addresses
- SATP register (CSR) includes:
 - Mode (e.g., Sv32, Sv39)
 - PPN of Base (a.k.a., “root”) PT
- A RISC-V Page Table Entry (PTE) includes:
 - Valid (V), Accessed (A) and Dirty (D) bits
 - Access Permissions (RWX)
 - RWX=000 → Pointer to next level PTE
 - Other metadata (User Mode, Global Mapping, reserved)



To Summarize: Cache vs. VM Terminology

- In caches, we dealt with individual **blocks**
 - Usually **~64B** on modern systems
- In VM, we deal with individual **pages**
 - Usually **~4 KB** on modern systems
- **Common point of confusion:**
 - **Bytes, Words, Blocks, Pages**
 - Are all just different ways of looking at memory!
- **Example for RV32:**
 - **1 GB** DRAM
 - **4 kB** pages
 - **64B** cache blocks

	<u>Cache</u>	<u>Virtual Memory</u>
Unit	Block or Line	Page
Hit/Miss	Miss	Page Fault
Unit Size	32-64B	4K-8KB
Placement	Direct Mapped, Set Associative	Fully Associative
Replacement	LRU or Random	LRU
Write Policy	Write Through or Write Back	Write Back



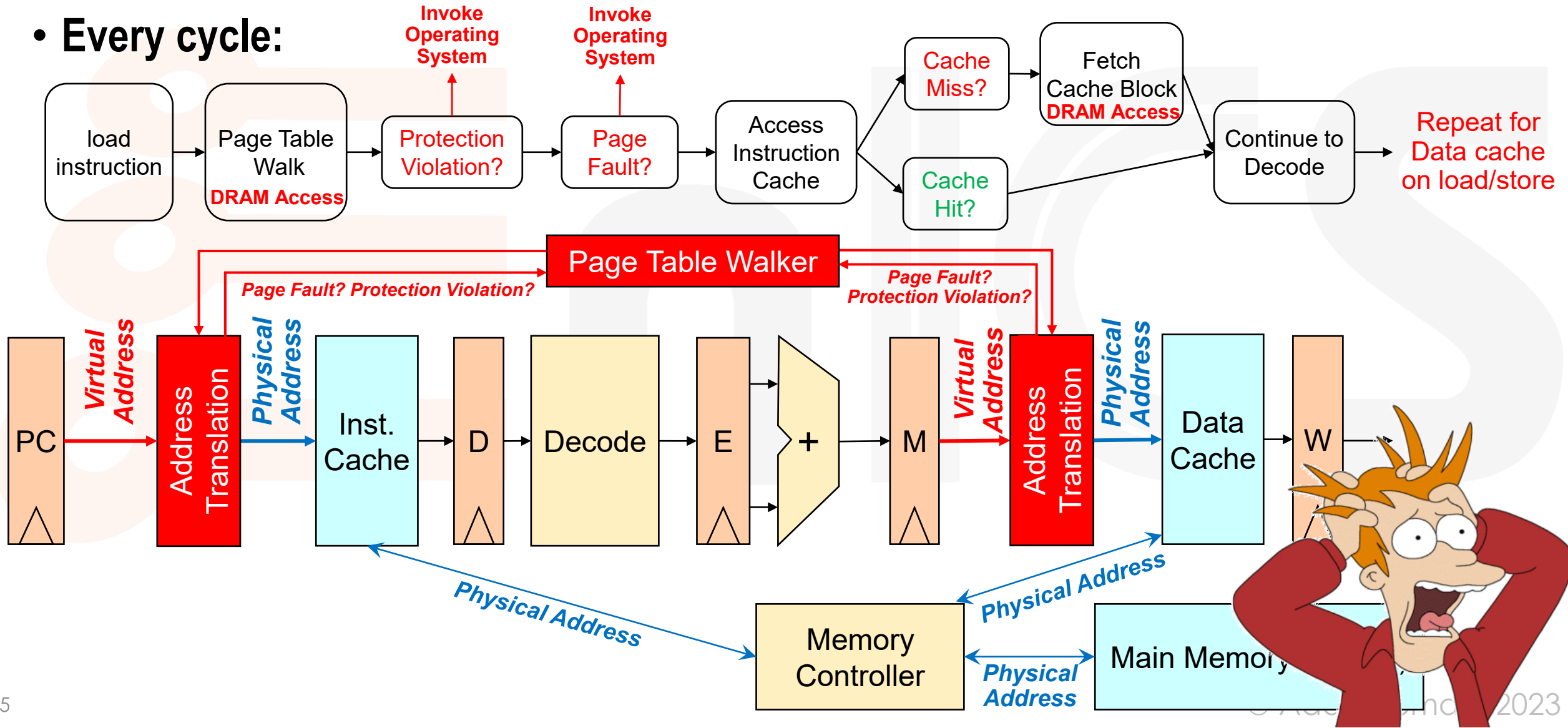


The Translation Lookaside Buffer (TLB)



Page-Based Virtual-Memory Machine

- Every cycle:

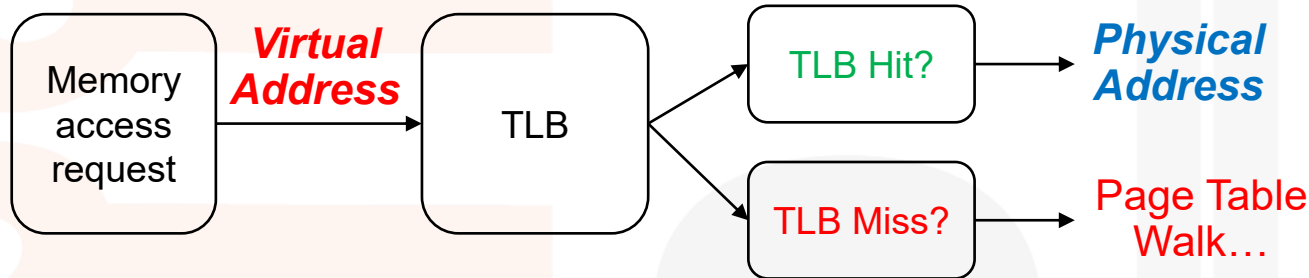


Virtual Memory Bottleneck

- **Virtual Memory access is expensive**
 - In a **single-level page table**, each reference becomes **two memory accesses**
 - In a **two-level page table**, each reference becomes **three memory accesses**
 - **We just totally killed our performance (CPI)...**
- **Any suggestions?**
 - Isn't *indirection* supposed to solve any problem in computer architecture?
 - Well, not this time... but wasn't there a "**second solution**"?
- **Caching, of course!**
 - Let's just cache our address translation.
 - If we store our recent VA-to-PA mappings on-chip, we'll save DRAM accesses.
- **This cache is called a **Translation Lookaside Buffer (TLB)****

Translation Lookaside Buffer

- A **TLB** is a **cache** of virtual-to-physical **address translations**

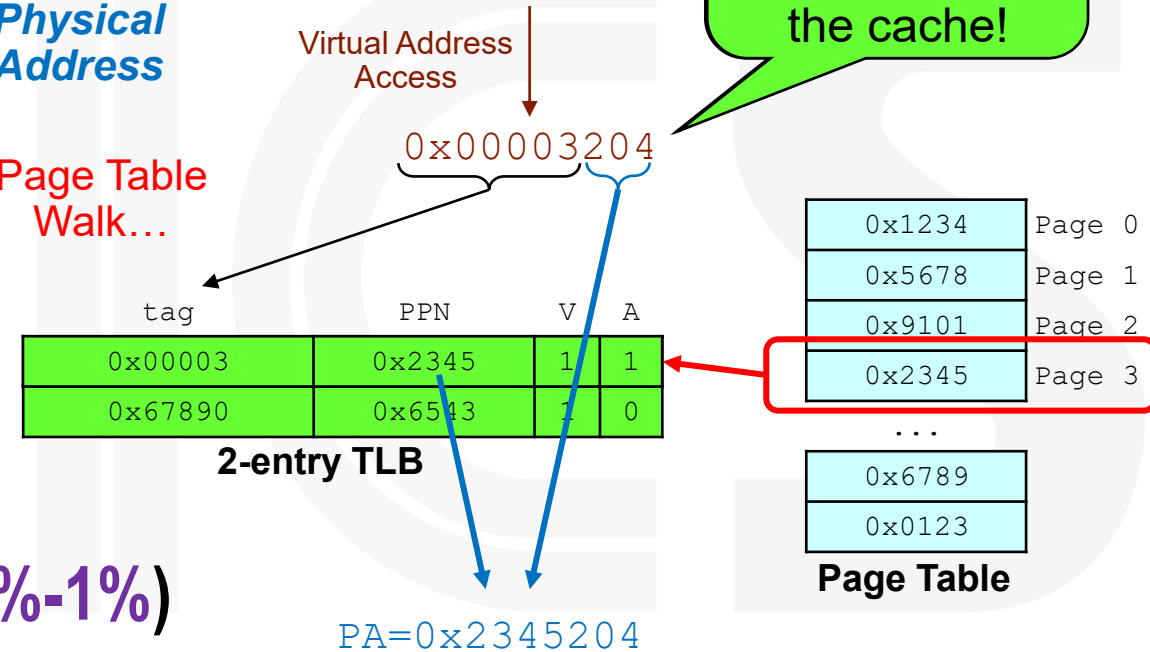


- TLBs need to be **fast** (→ **small**...).

- Luckily, a small TLB is **very effective**!
- Typically, **16-512 entries**.

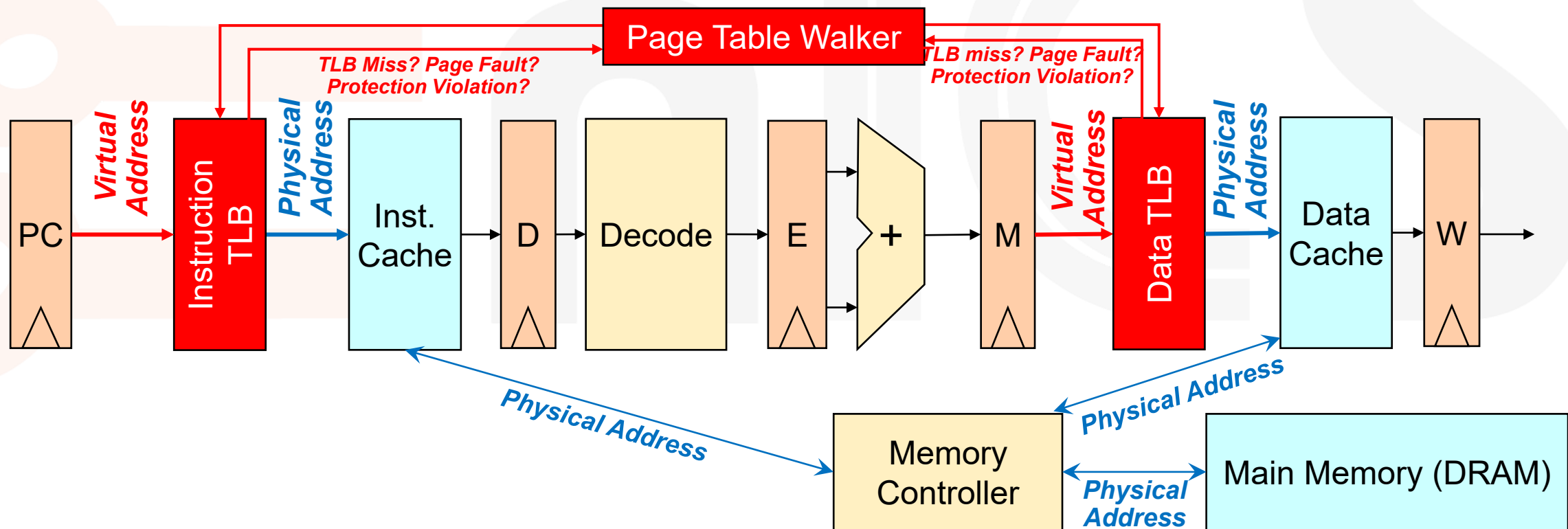
- Minimizing **TLB miss rate** is critical (→ **0.01%-1%**)

- Usually **4-way/8-way/fully-associative**.
- **Random** or **FIFO** replacement policy, since **LRU** too expensive.
- A **2nd-level TLB** can be bigger (and slower) to **increase hit rate**.
- **Larger page sizes** also reduce TLB miss rate.



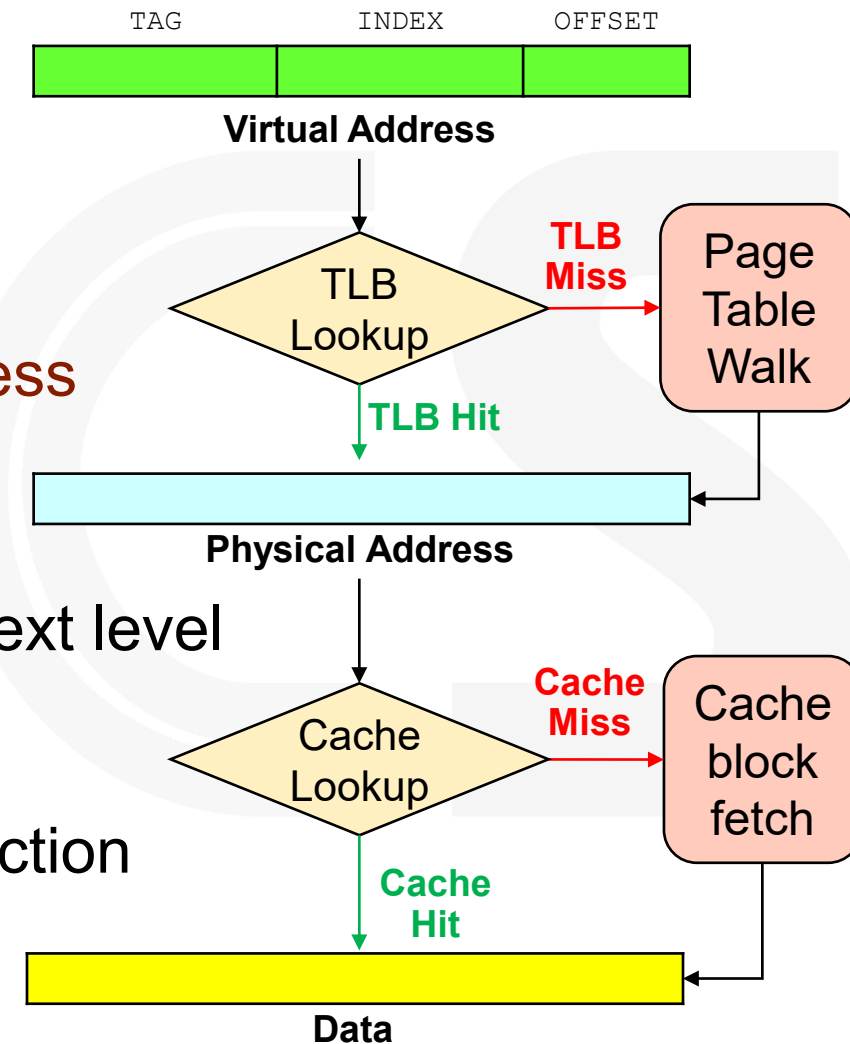
Processor Pipeline with TLB

- So, now we have added two TLBs in the pipeline:
 - One for Instruction Memory Access
 - One for Data Memory Access



Physically-Indexed Physically-Tagged

- The pipeline from the previous slide is known as a **physically-indexed physically-tagged (PIPT)** cache.
 - CPU generates **virtual address**
 - A **TLB-lookup** is performed to get the **physical address**
 - A **TLB miss** initiates a **page walk**
 - The physical address is used to access the **cache**
 - A **cache miss** requires fetching the block from the next level
- This process is **very slow**
 - Memory access happens several times every instruction
 - The TLB significantly slows down memory access
- Can we make it better?

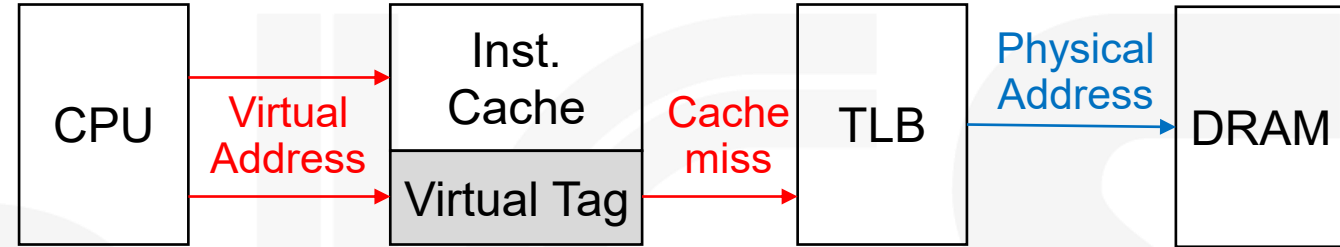


Virtually-Addressed Caches

- **Solution: Access the cache with a **virtual address**!**

- Now we only have a **TLB lookup** if we have a **cache miss**!

- This type of cache is called **Virtually-Indexed Virtually Tagged (VIVT)**



- **But does it work?**

- **Unfortunately, not really...**

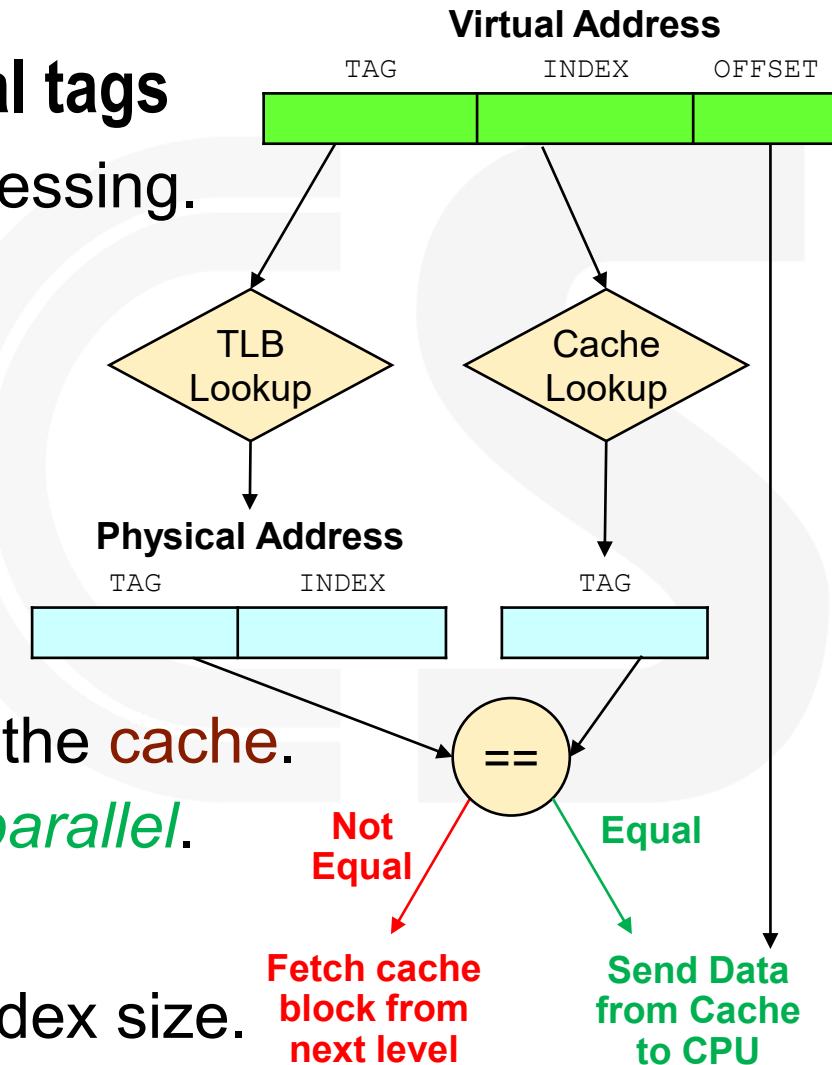
- Permissions are stored in the TLB → There's no **protection**.
- Different processes, Same VA (**Homonym**) → **Flush the cache** on context switch.
- Two VAs for same physical address → **Aliasing** – stale data in cache.



- Any way to get “**the best of both worlds**”?

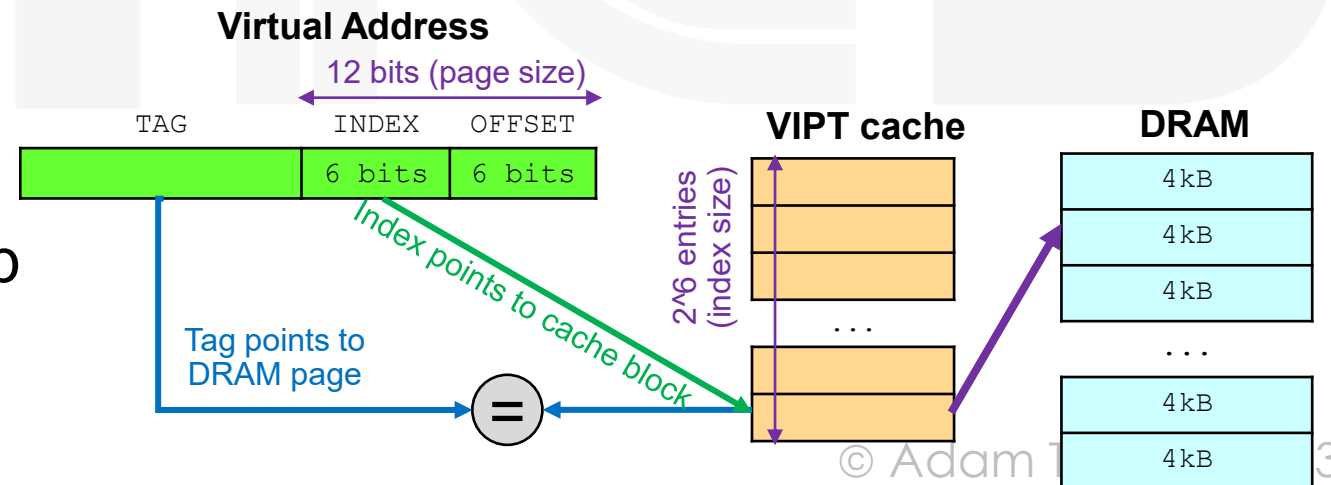
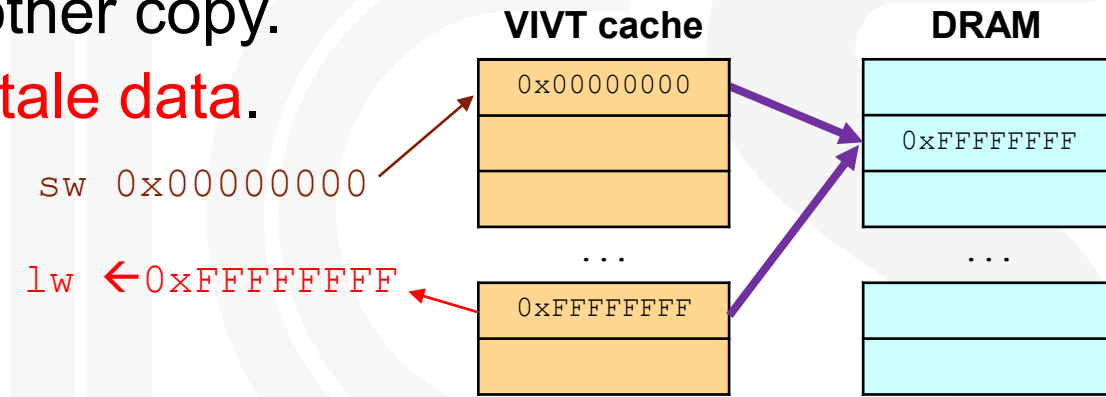
Virtually-Indexed Physically-Tagged Cache

- So, we understood that we almost *have to* use physical tags
 - Physical tags enable **process isolation** of virtual addressing.
- But the TLB slows our processor down.
 - Could we possibly “hide” the TLB lookup?
- Introducing the **Virtually-Indexed, Physically-Tagged (VIPT) Cache**
 - Always access the **physically-tagged TLB**.
 - But use the **index bits** of the address to store data in the **cache**.
 - This way, we can perform **TLB** and **cache lookup** *in parallel*.
- This is the way most L1 caches are built today
 - However, note that the **cache size is limited** by the index size.
 - Using a **larger cache tag** than the index will lead to **aliasing**.



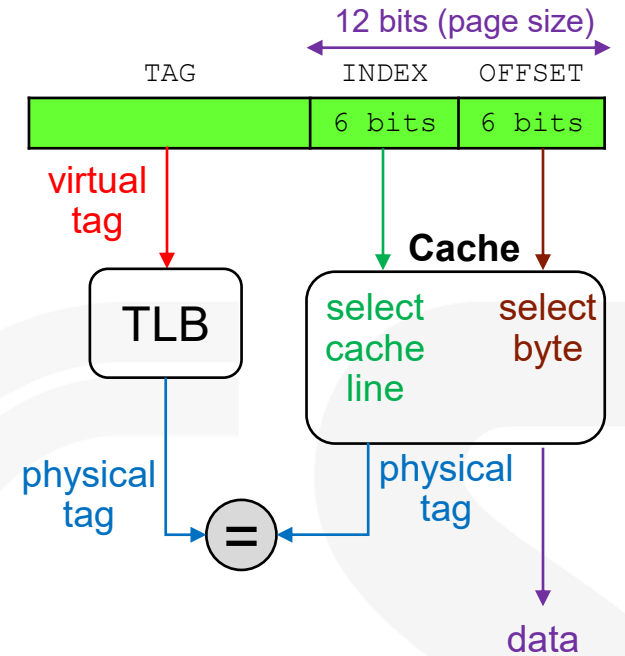
Overcoming Aliasing in VIPT Caches

- When using a **VIVT cache**, what happens when **two VAs point to the same PA**?
 - Two copies of **same physical address** can be brought into cache.
 - Writing to one copy is **not reflected** in the other copy.
 - Reading the second copy would result in **stale data**.
- **VIPT caches** overcome this by storing and comparing the **physical tag**
 - **DRAM pages** are larger than **cache lines**.
 - So, we can use the **index bits** to address the cache independent of the **tag**, which can be **physical**!
 - Therefore, we can do a TLB lookup **in parallel** to a cache access.



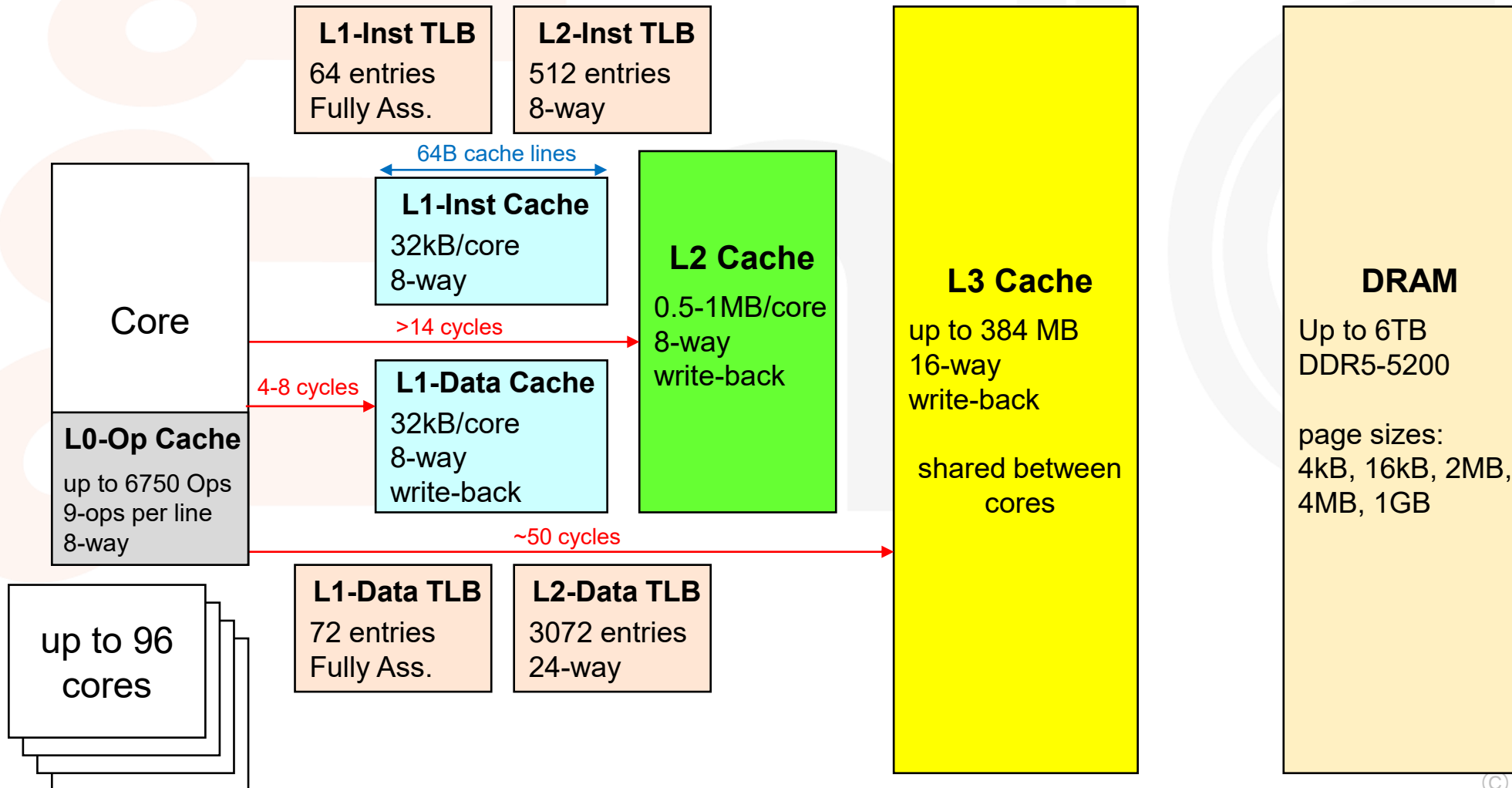
Limitations of VIPT Caches

- What is the maximum size of a VIPT Cache?
 - Assume 4kB DRAM pages and 64B cache lines.
 - Page offset is 12 bits and cache block offset is 6 bits.
 - If the index overlaps the page offset, part of the index requires translation and therefore we get aliasing
 - That leaves us 6 bits for the index (in a direct-mapped cache).
 - → Cache size (without aliasing) is ≤ 64 entries (4kB)
- Increasing associativity enables larger caches
 - 2-way associativity means that for a given index, the tag could be in two places.
 - So, in the example, a 6-bit index enables $2^6=64$ entries (8kB)
 - That's why we find high-associativity (~ 16 -way) in inner-level caches!



Example: AMD Ryzen 7000 (zen4)

- To give a recent example, let's look at the zen4 architecture (Nov. 2022)



Source: wikichip

© Adam Teman, 2023

Virtual Memory Summary

- **Virtual memory adds a level of **indirection** between the program and the memory**
 - Enables us to provide “unlimited memory” to each process
 - Isolates programs (full address space, protection, data sharing)
 - Eliminates fragmentation
- **However, accessing memory using VM is expensive**
 - First, need to access the **page table** find the physical address.
 - Then need to **access again** to retrieve data from DRAM.
 - If page not resident in memory, **page faults** are really bad.
- **Make it faster by caching VA to PA translations – use a TLB.**
 - **PIPT Cache** – requires translation first → slow
 - **VIVT Cache** – no translation → fast, but really impractical
 - **VIPT Cache** – parallelize TLB and Cache access → fast, but cache size limited

References

- **Patterson, Hennessy “Computer Organization and Design – The RISC-V Edition”**
- **Hennessy, Patterson “Computer Architecture – A Quantitative Approach”**
- **Krste Asanovich, Berkeley 61C**
- **MIT 6.175**
- **Chris Terman, MIT 6.004**
- **Georgia Tech CS6290 “High Performance Computer Architecture” – available on Udacity**
- **David Black Schaffer “Virtual Memory” <https://youtu.be/qcBlvnQt0Bw>**