Caches and Virtual Memory

Computer Architecture
ECE 6913

Brandon Reagen

Announcements

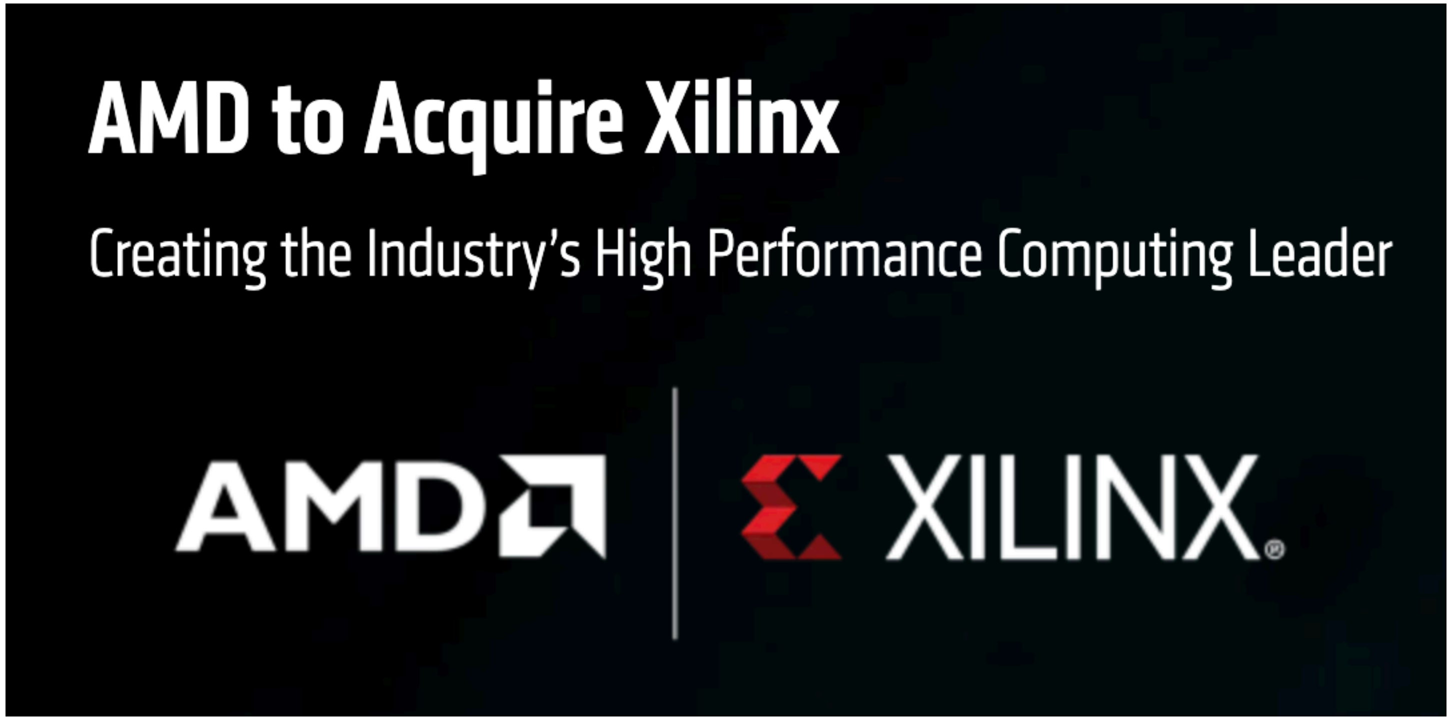
1) Exam
    1) How was it?
    2) TAs are grading now. Will release with grading system

2) Lab2
    1) How's it going?
    2) Please check-in with TAs with any questions

3) Today
    1) Finish caches
    2) Start virtual memory

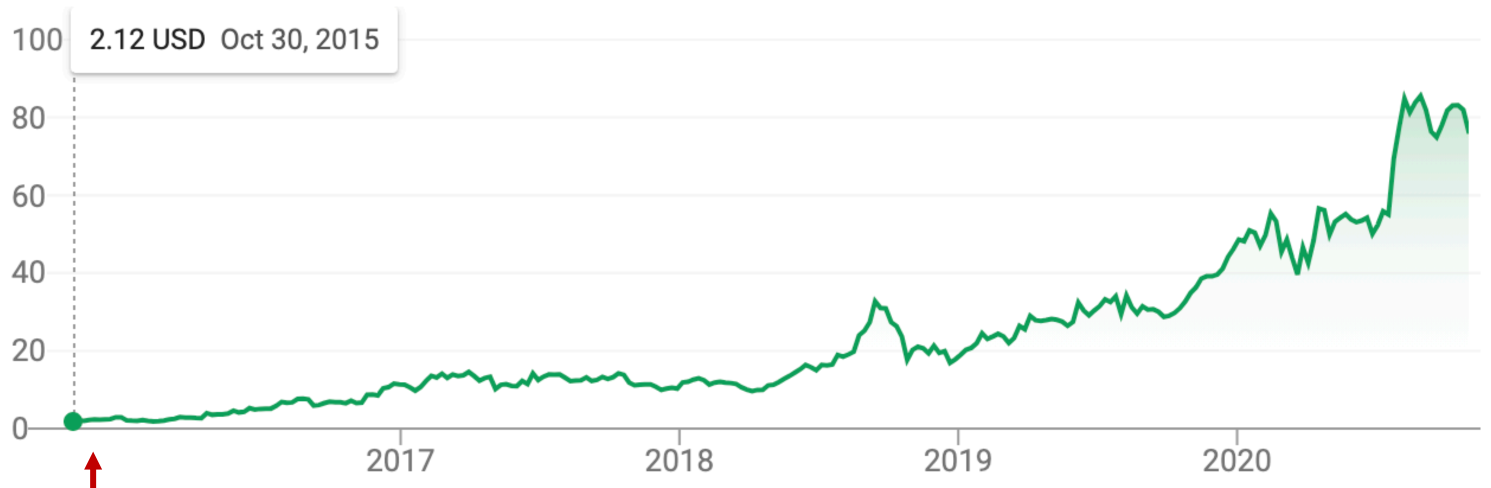
What happened this week?

What happened this week?


AMD to Acquire Xilinx

Creating the Industry's High Performance Computing Leader

AMD's amazing turn around..



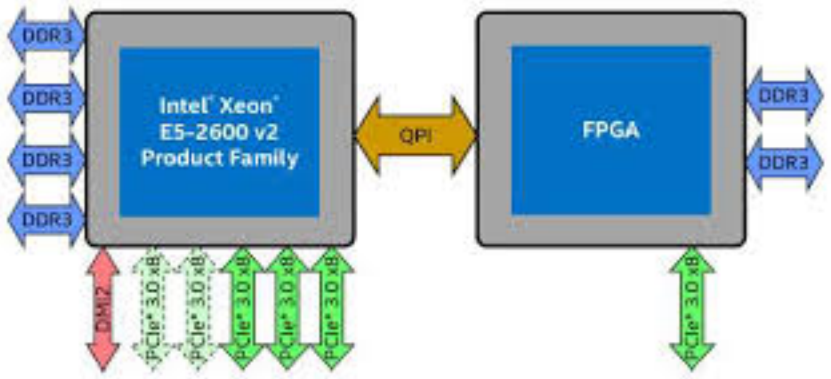
When I interned at AMD

Price '15: $1.8
Price today: $75
42x!

Today cap: 90B
2015 cap: ~2B

Now buying Xilinx for $35B

Why's this exciting?



Intel HARP system

Starting to tightly integrate FPGAs and CPUs through LLC.
Supporting hardware customization for improved perf/efficiency

Why is specialization so important?
    Allows you to customize circuits to do 1-2 things very well
    elide costs of general-purpose processors

Having this system presents a lot of opportunities for
    radically new computing systems

What happened this week?

Cache Organization

Caches store data in blocks

- A block is a collection of contiguous bytes
  - Can be as small as 1 byte, but can be larger (example: 128 bytes)
  - What's the size of blocks today?
- Request to a cache operate on an entire cache block

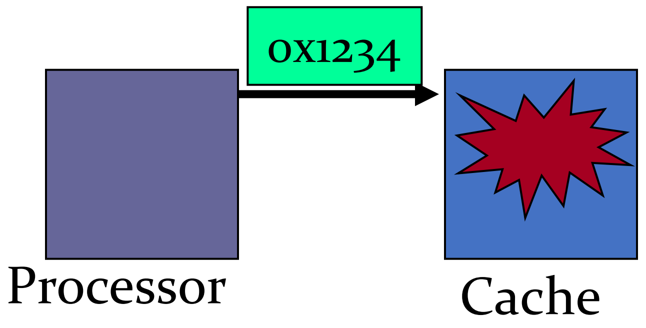
Key questions we need to answer:

- Placement: Where does a block go when it is fetched into cache?
- Identification: How do we know if a block already exists in the cache?
- Replacement: Which block should we kick out if there isn't enough room?

Three Cs (Cache Miss Terms)

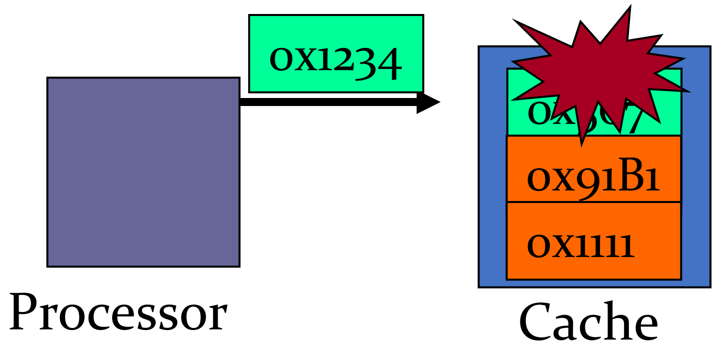
Compulsory Misses:

- "Cold start" misses
- Caches do not have valid data at the start of the program



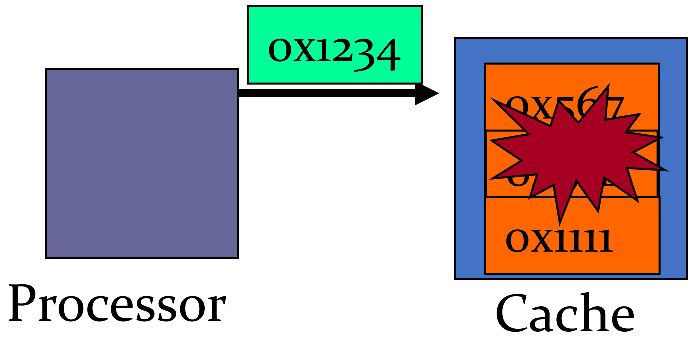
Conflict Misses:

- Increase cache associativity
- What we saw with DM caches
- Associative caches reduce conflict misses



Capacity Misses:

- Increase cache size



Another way to think about it

Conflict misses:

Miss could be avoided with different hashing

This means: You missed because the set was full and had to
evict a block that will be used again

If you could have put the block <u>anywhere</u> in the cache,
the miss would be avoided
(empty slot or slot with data never used again)

Conflicts are caused by hashing addresses to sets,
this mapping changes with associativity and capacity

Capacity misses:

Can only be resolved by increasing capacity

You can evict anything you want, but all slots are full and all data needed again

Types of Caches

Type of cache	Placement: Mapping of data from memory to cache	Identification: Complexity of searching the cache
Direct mapped (DM)	ed at **ion**	Fast indexing mechanism
Set-associative (SA)	A ory value can be placed in **ny of a set of locations** in the cache	Slightly more involved search mechanism
Fully-associative (FA)	A memory value can be placed in **any location** in the cache	Extensive hardware resources required to search (CAM)

- DM and FA can be thought as special cases of SA
  - DM → 1-way SA
  - FA → All-way SA

NYU TANDON SCHOOL OF ENGINEERING

Think about placement for a minute..

Caches are temporary structures that store memory

We can’t address caches like memory!

Caches are addressed by searching for content
(i.e., data/instruction address)

Think about a looking for a book in a library (ordered)
verses checking the return pile (who knows!)

Set Associative Caches

Direct mapped caches can have high miss rates due to conflicts

- Each address maps to a unique location in the cache
- What's this mean? Poor utilization!

Assume addresses A and B with same index bits

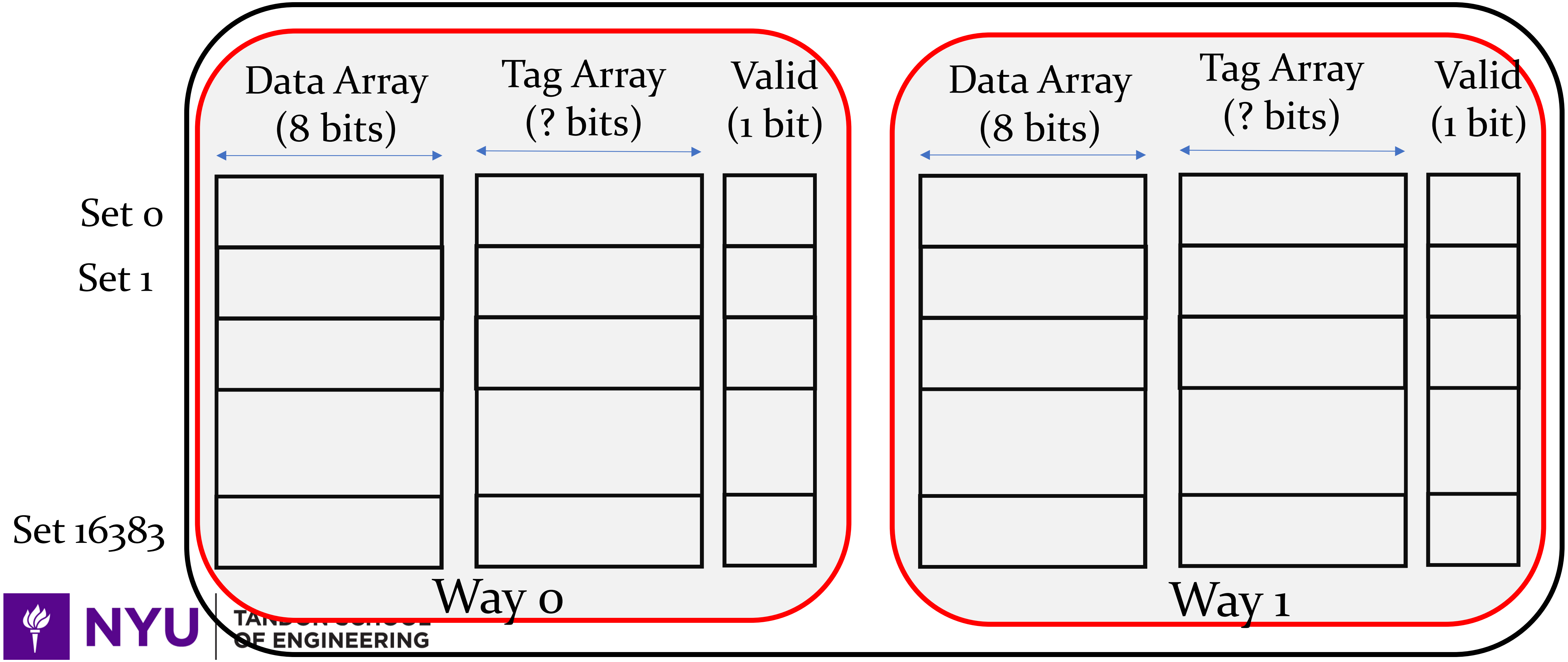
- Sequence: A, B, A, B, A….. results in 100% cache miss rate!

Set-associative cache: each address can map to
N different locations (ways) in the cache!

- "N-way" set associative cache
- 2-way set associate cache has
  ~0% cache miss rate for sequence A, B, A, B …

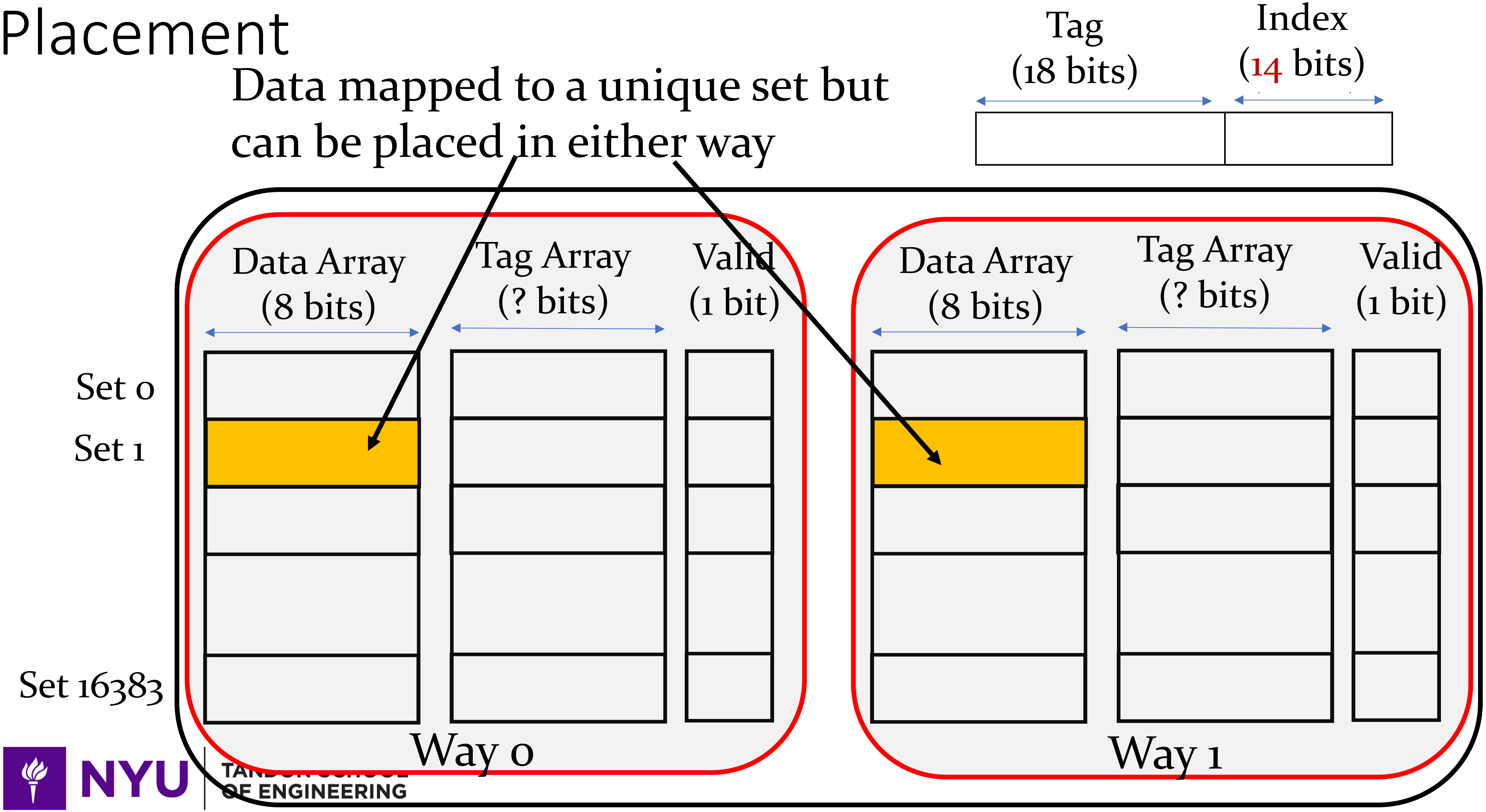
2-Way Set Associative Cache

2-way set-associative 32 KB cache with 1 Byte blocks



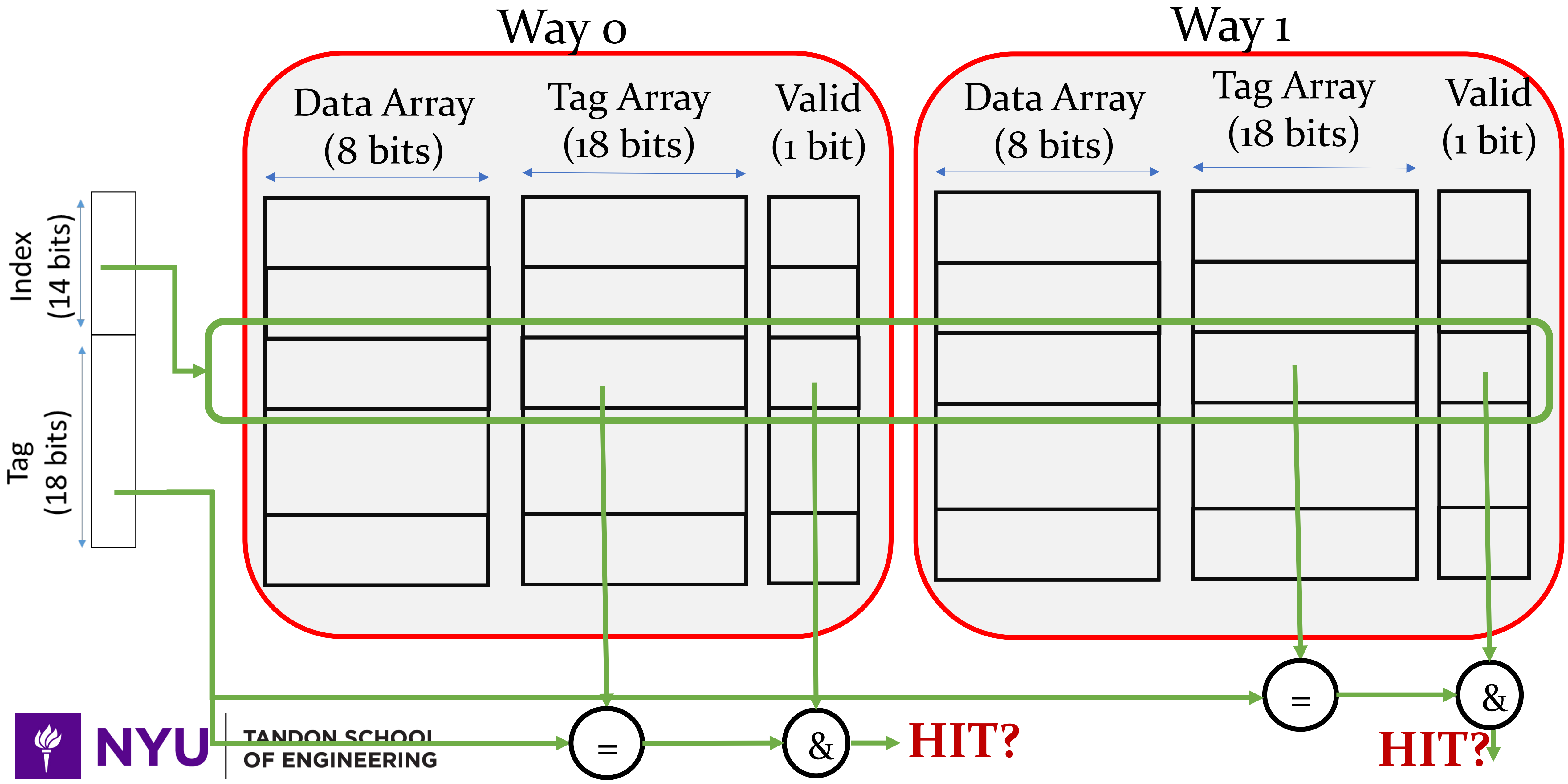
Placement

Data mapped to a unique set but can be placed in either way



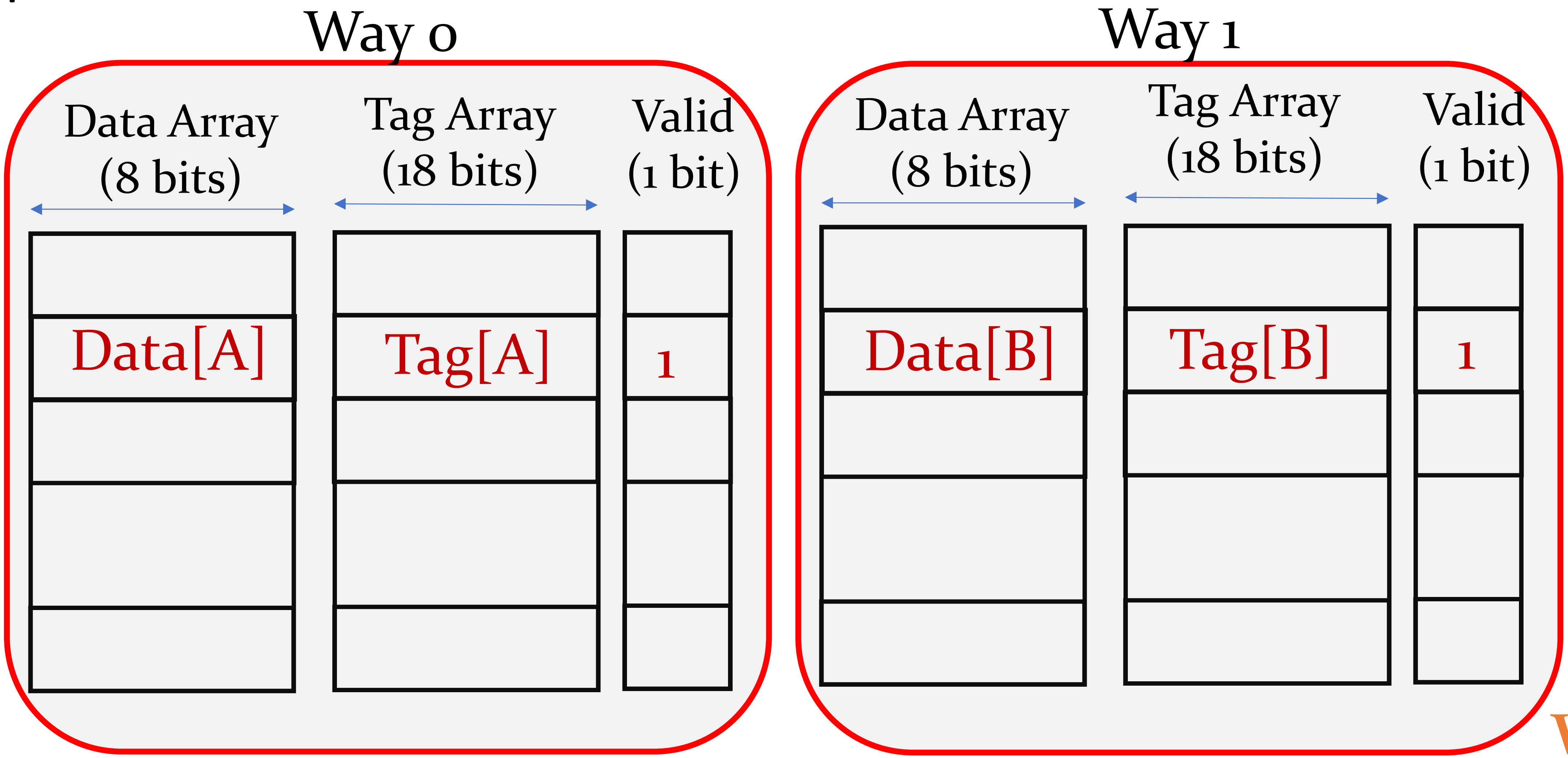
NYU TANDON SCHOOL OF ENGINEERING

Identification



Replacement

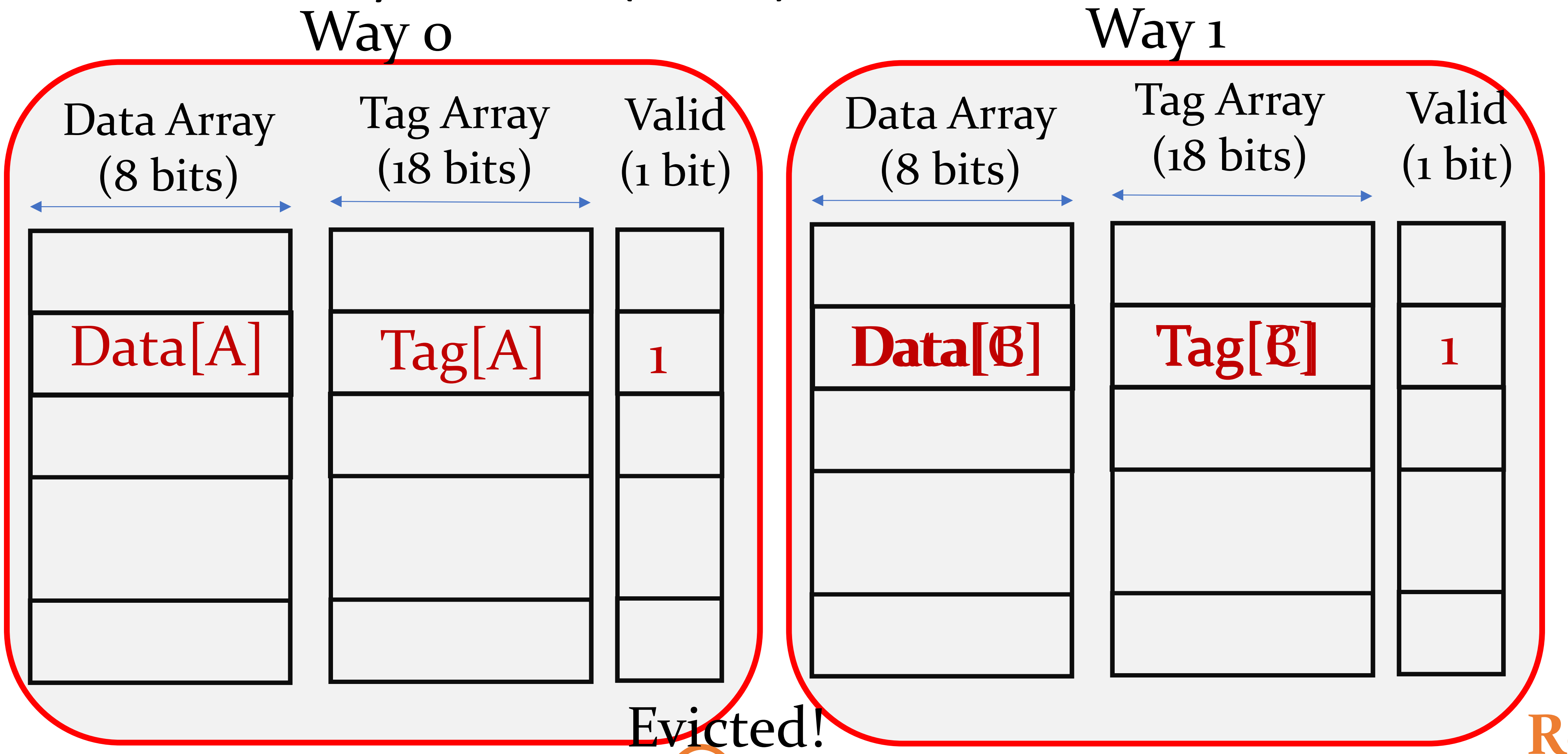
(Addresses A, B, C map to same set)



Sequence:	A	B	B	A	A	A	A	C
	M	M	H	H	H	H	H	M

Where should C go?

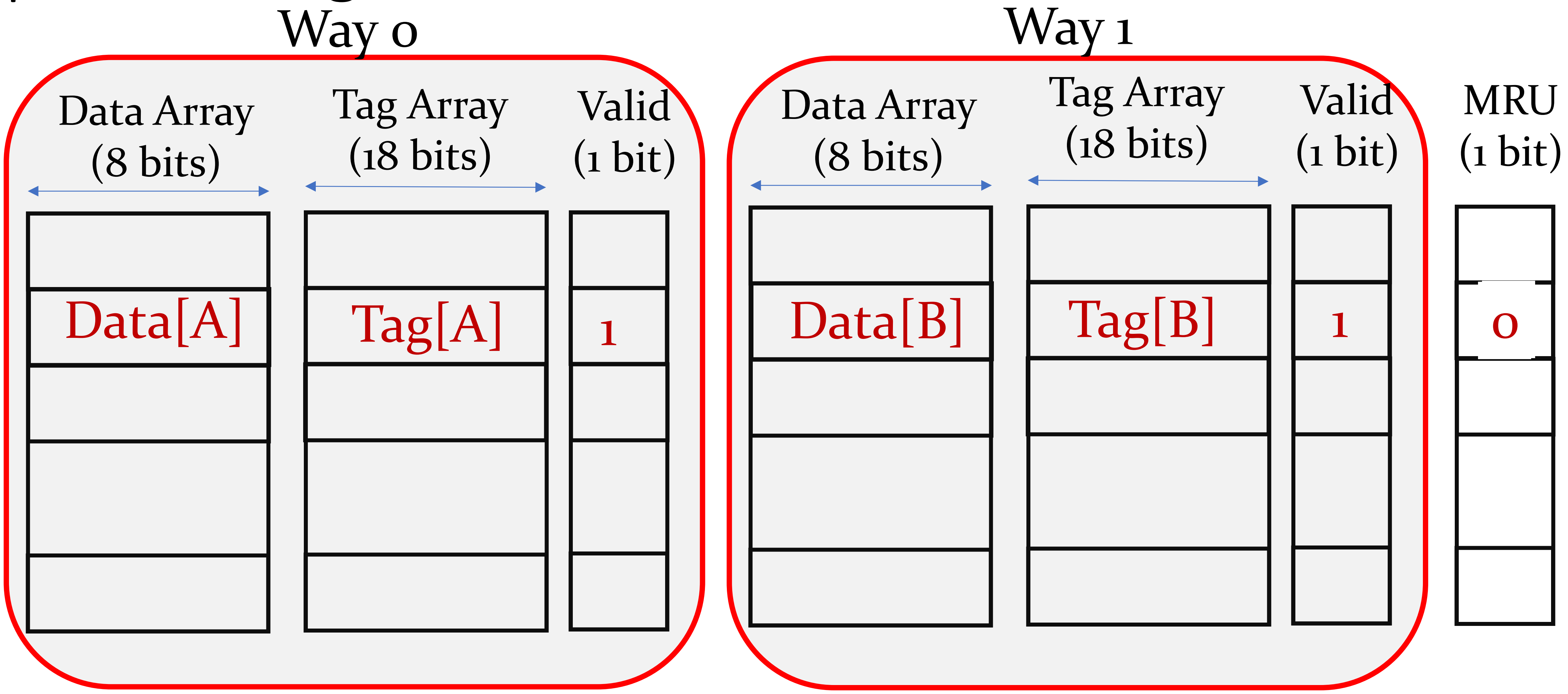
Least Recently Used (LRU)



Sequence:	A	B	B	A	A	A	A	C
	M	M	H	H	H	H	H	M

Replace least recently used block

Implementing LRU



Sequence:	A	B	B	A	A	A	A	C
	M	M	H	H	H	H	H	M

LRU = NOT MRU

Practical LRU Implementation

LRU is hard to implement in hardware when N>2

- Keep track of all possible N! orderings of N ways
- A linked list in which the head points to MRU and tail points to LRU
- Example: N=4; 4 x 2 bits = 8 bits per cache set and extra logic to update list on every access

2b per line or more lines?

Random isn't all that bad..

Intuition: LRU is an approximation anyways.. What's that mean?
Might not be the optimal replacement policy!

Alternative policies that are more hardware friendly

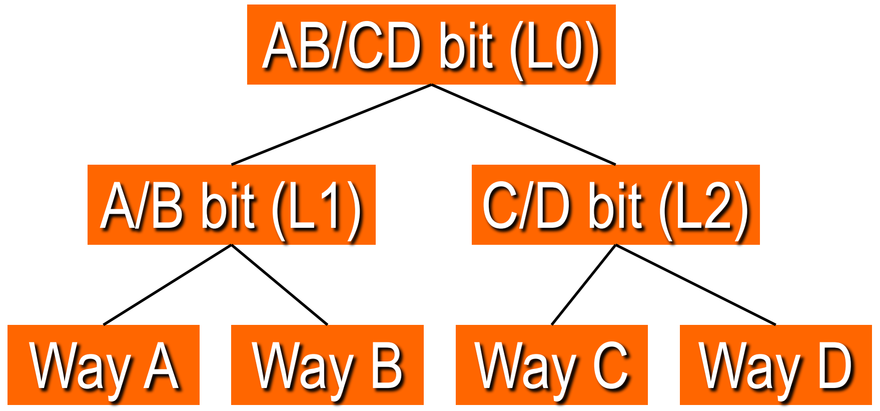
- NOT MRU: same as LRU for N=2, requires only log(N) bits, easy update
- Hierarchical: for N=4, divide ways into 2 groups of 2 ways

Hierarchical LRU example

	Group 0		Group 1	
Start	A 11	B 10	C 01	D 00
Access to B	A 10	B 11	C 01	D 00
Access to D	A 00	B 01	C 10	D 11
Access to X	X 11	B 10	C 00	D 01

What locality is this exploiting?

Pseudo LRU Algorithm (4-way SA)

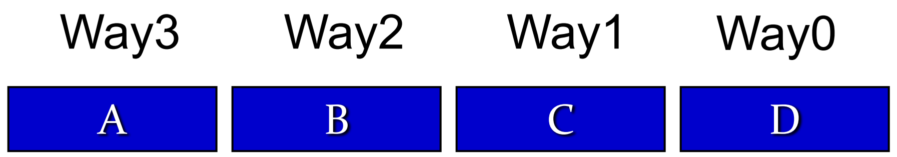


Tree-based

O(N): 3 bits for 4-way

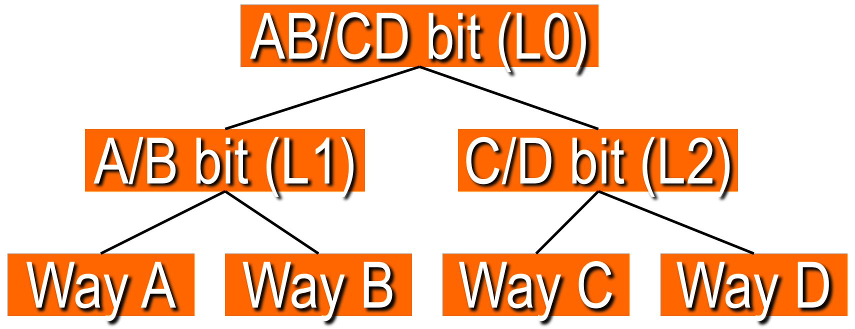
Cache ways are the leaves of the tree

Combine ways as we proceed towards the root of the tree



NYU TANDON SCHOOL OF ENGINEERING

Pseudo LRU Algorithm



Less hardware than LRU

Faster than LRU

Assume here: 0 goes left, 1 goes right

Ones point to what should be removed

•L2L1L0 = 000, there is a hit in Way B, what is the new updated L2L1L0?

•L2L1L0 = 001, a way needs to be replaced, which way would be chosen?

LRU update algorithm

	CD	AB	AB/CD
Way hit	L2	L1	L0
Way A	---	1	1
Way B	---	0	1
Way C	1	---	0
Way D	0	---	0

Replacement Decision

CD	AB	AB/CD	
L2	L1	L0	Way to replace
X	0	0	Way A
X	1	0	Way B
1	X	1	Way C
0	X	1	Way D

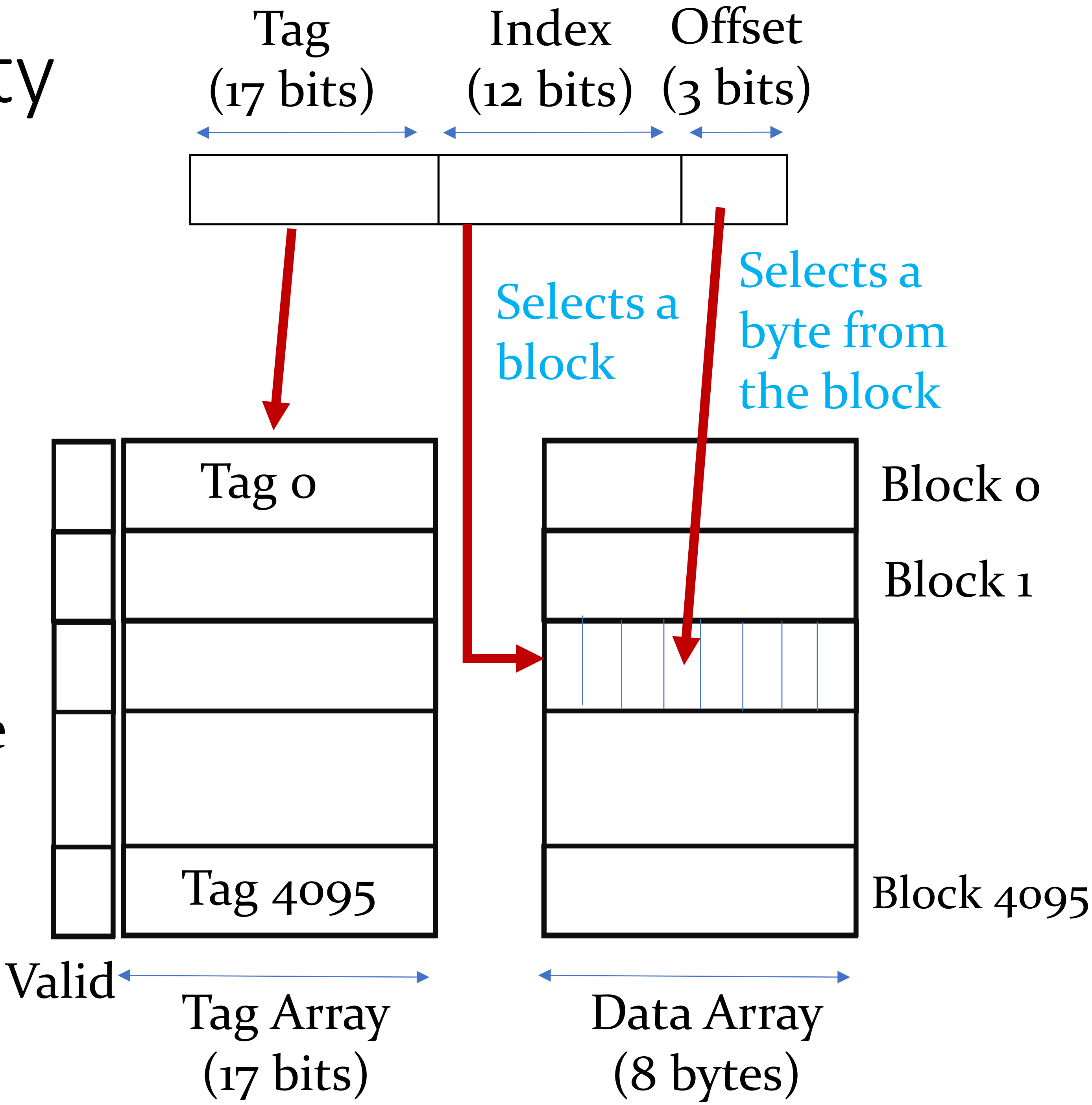
Exploiting Spatial Locality

Recall that if the byte from address i is accessed, then byte from address i+1 is likely to be accessed

- Pull in *multiple* contiguous bytes of data in each access
- Use larger block size!

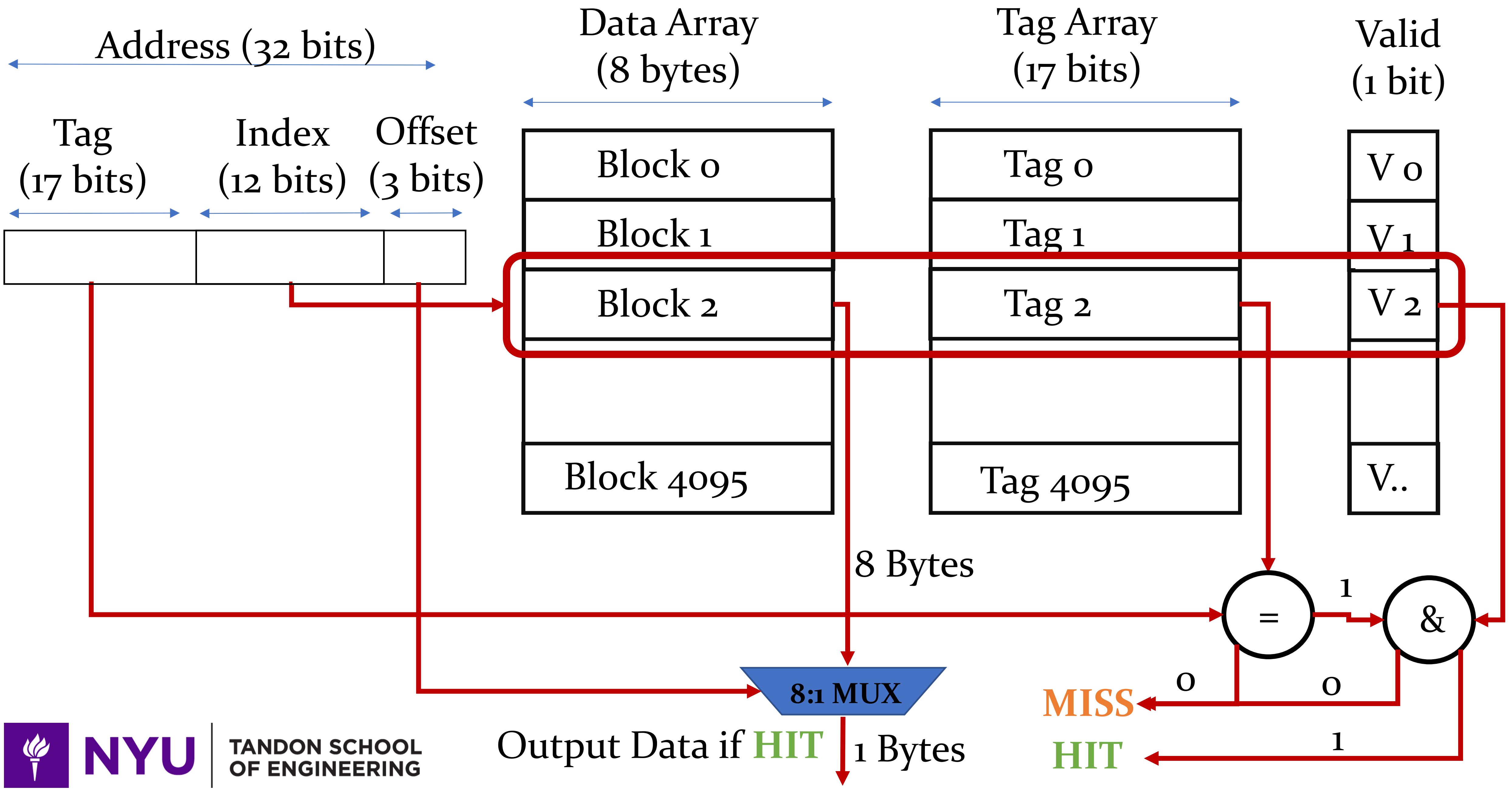
Example: 32KB direct mapped cache with 8 Byte (64 bit) blocks

- i.e., cache has 4096 blocks



Cache Operation

What type of cache is this?



NYU TANDON SCHOOL OF ENGINEERING

Impact of Block Size

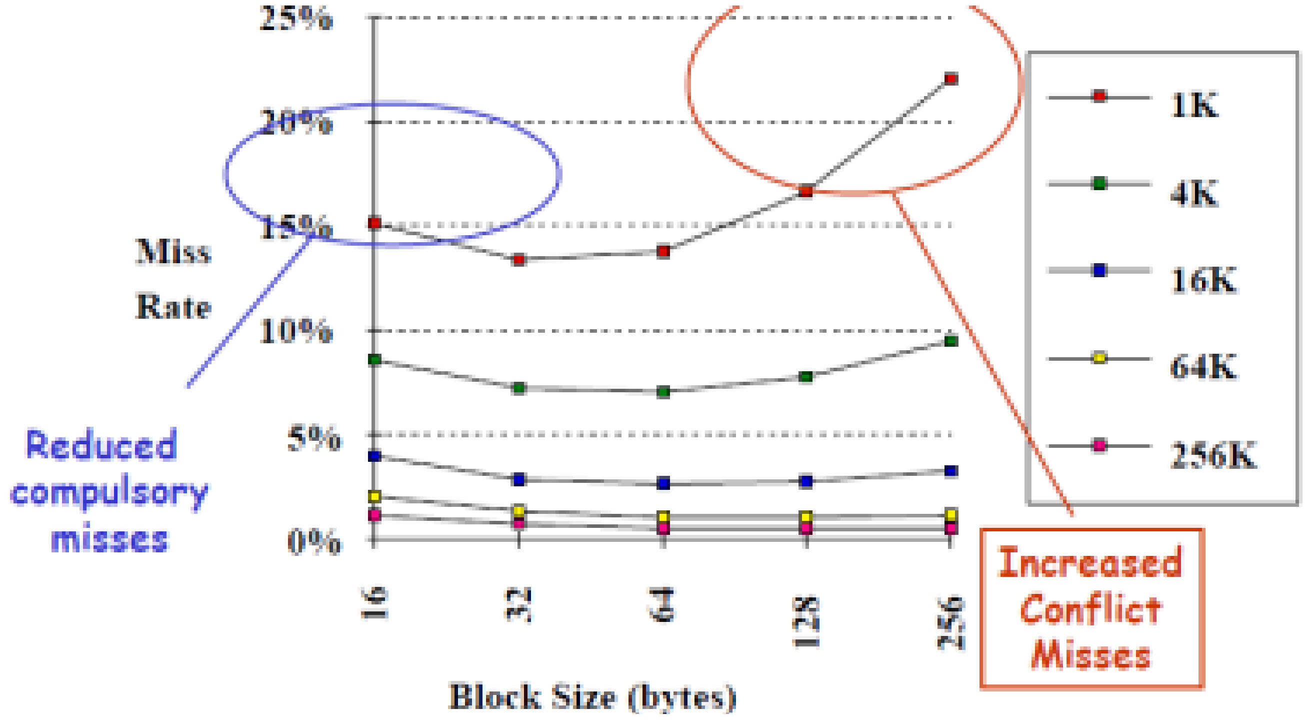
Sequence of addresses: A, A+1, A+2, A+3 …

- 4 consecutive misses for 1 byte block size
- 1 miss and 3 hits for 4 byte block size

Small block sizes don't exploit any spatial locality

What happens if the block size increases for the same cache size

- Fewer number of larger blocks



Write Policies

What should we do on a store/write event?

- Cannot perform tag look-up and write to the data array in parallel (why?)
- First access tag array and if there is a write hit, write to the data array
- Increases the delay of a cache access
  (recall: period is determined by the worst-case)

What do on a write hit?

- When there is a tag match (i.e., block exists in cache)

What to do on a write miss?

- When the data block is not in the cache?

Write Hit Policies

When to propagate new (“dirty”) values to lower levels

- Write-back policy: lazy, take care of it later
- Write-through policy: update lower levels immediately

Write-back policy

- Modify the data in the current cache level only
- When to update the data in the lower level? When cache block is evicted
- Dirty bit per cache block to keep track of blocks that have been updated

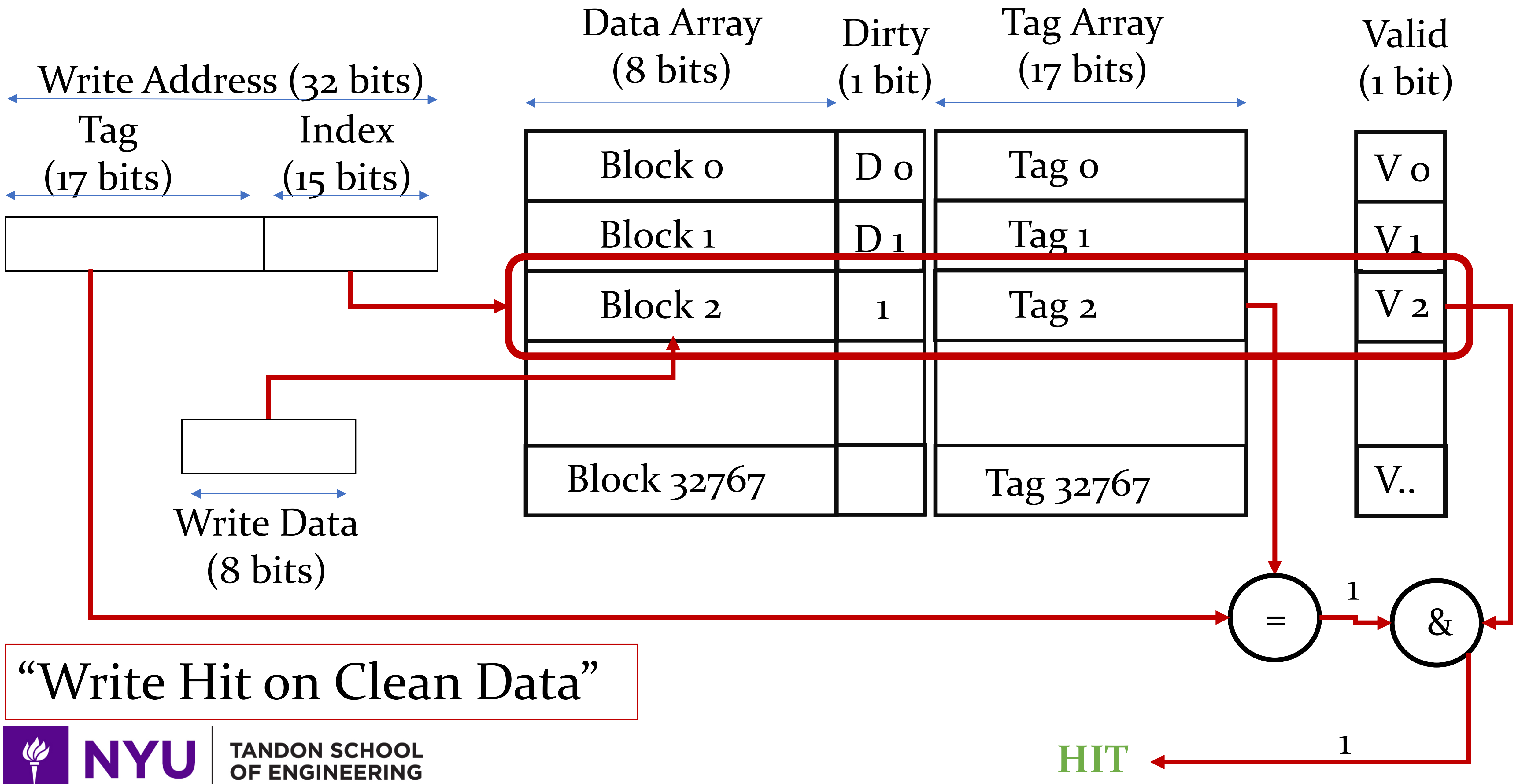
Pros

- Write happens at speed of current cache level
- Multiple writes to the same block
  result in only one write back to main memory

Cons

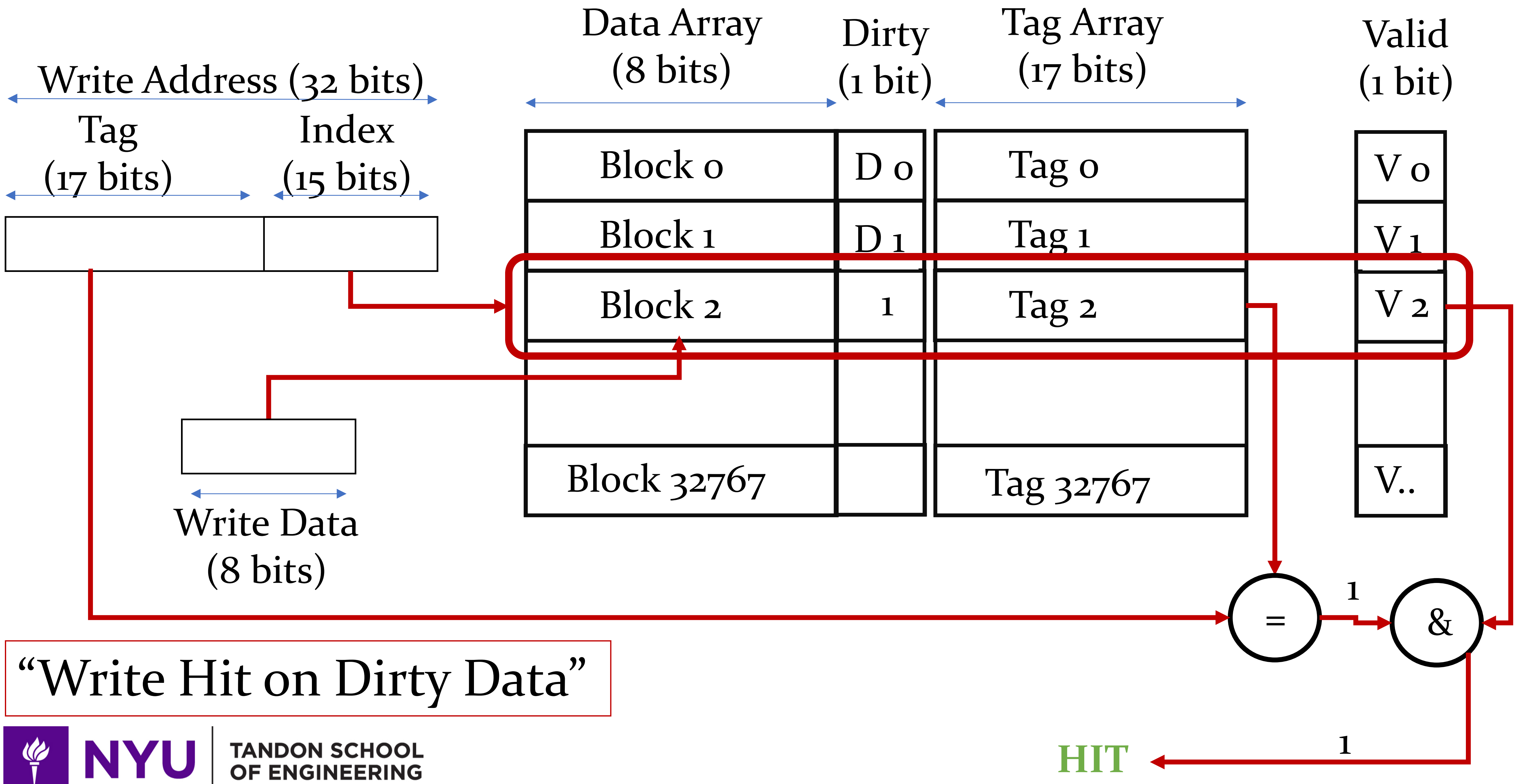
- Evictions take more time
- Data inconsistency between cache and lower levels

Write Back Cache (32 KB, Direct Mapped, 1 Byte Block)



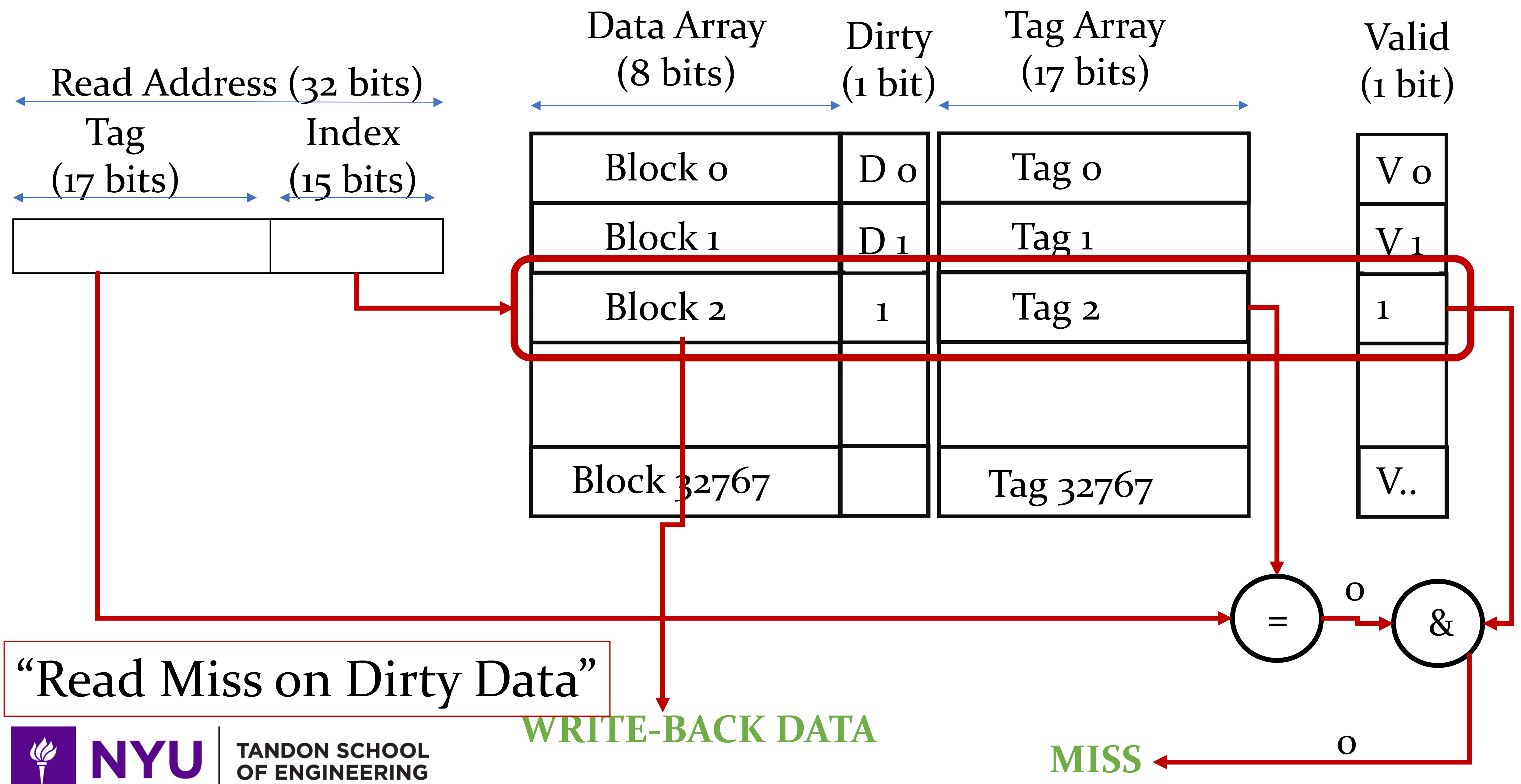
"Write Hit on Clean Data"

Write Back Cache (32 KB, Direct Mapped, 1 Byte Block)



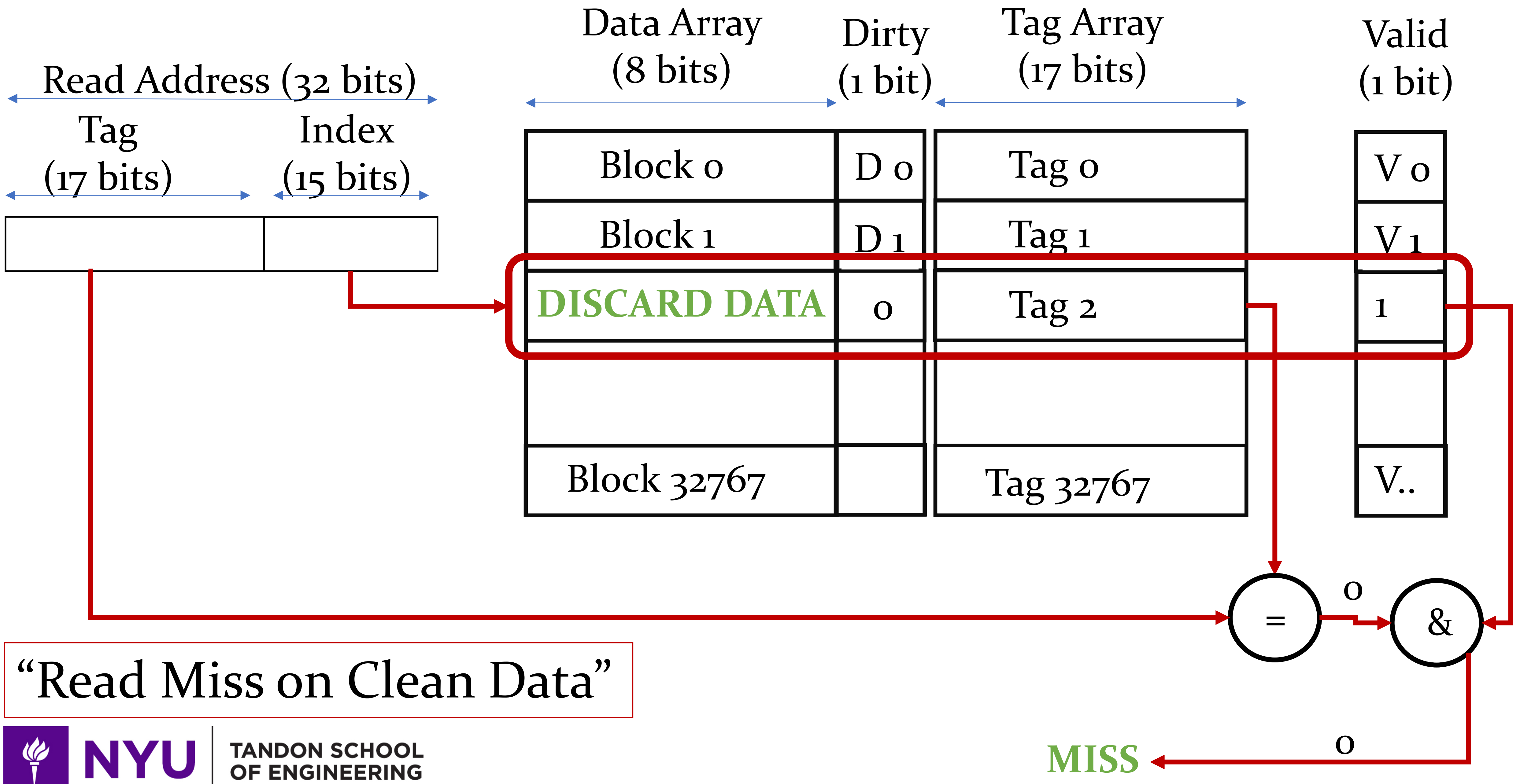
"Write Hit on Dirty Data"

Write Back Cache (32 KB, Direct Mapped, 1 Byte Block)



"Read Miss on Dirty Data"

Write Back Cache (32 KB, Direct Mapped, 1 Byte Block)



“Read Miss on Clean Data”

Write Hit Policies

When to propagate new ("dirty") values to lower levels

- Write-back policy: lazy, take care of it later
- Write-through policy: update lower levels immediately

Write-Through policy

- Update lower levels of cache/memory on every write
- No need for a dirty bit in the cache

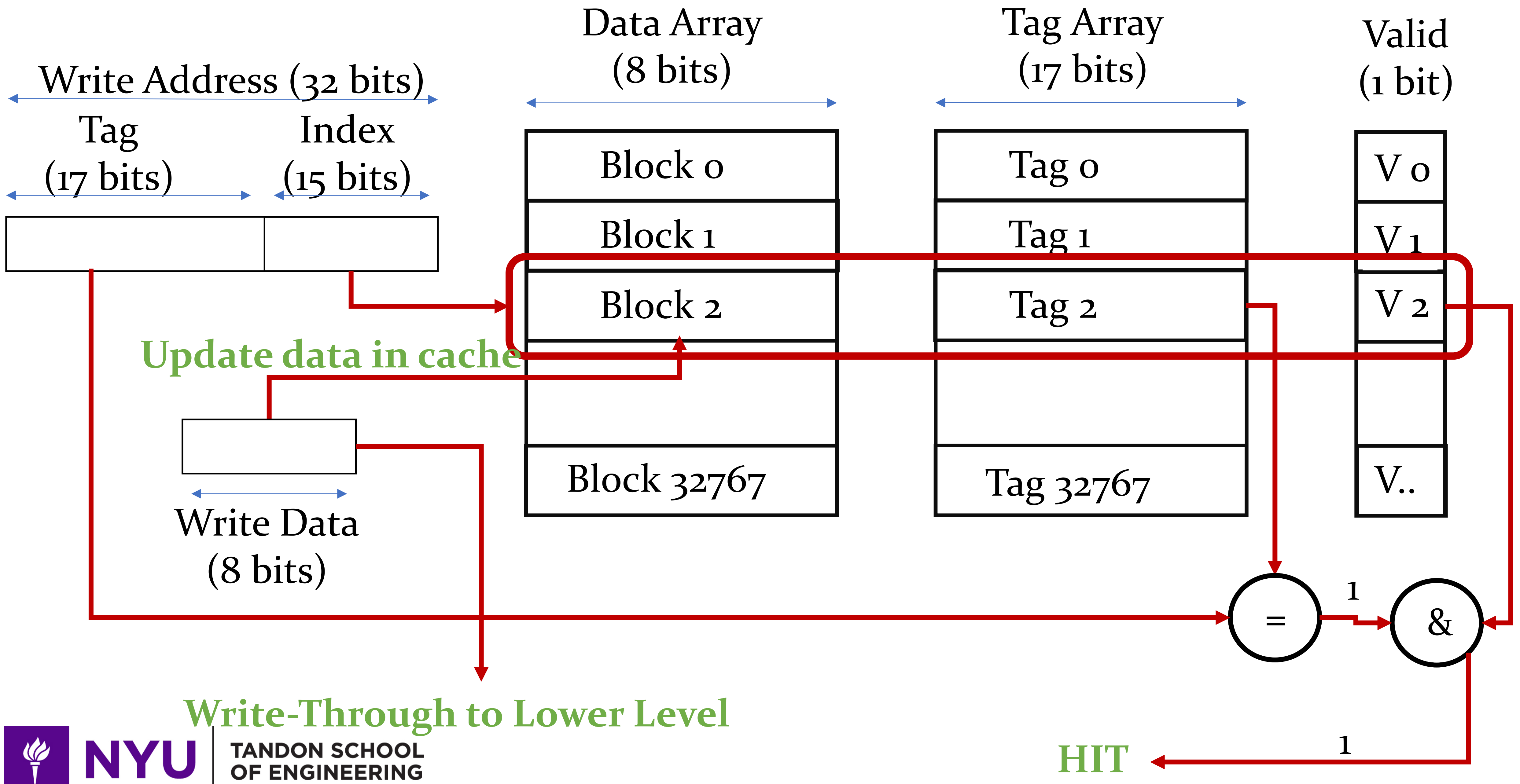
Pros

- Reduces complexity of cache (no dirty bit)
- Reads never cause write-backs
- Consistency across levels of memory hierarchy

Cons

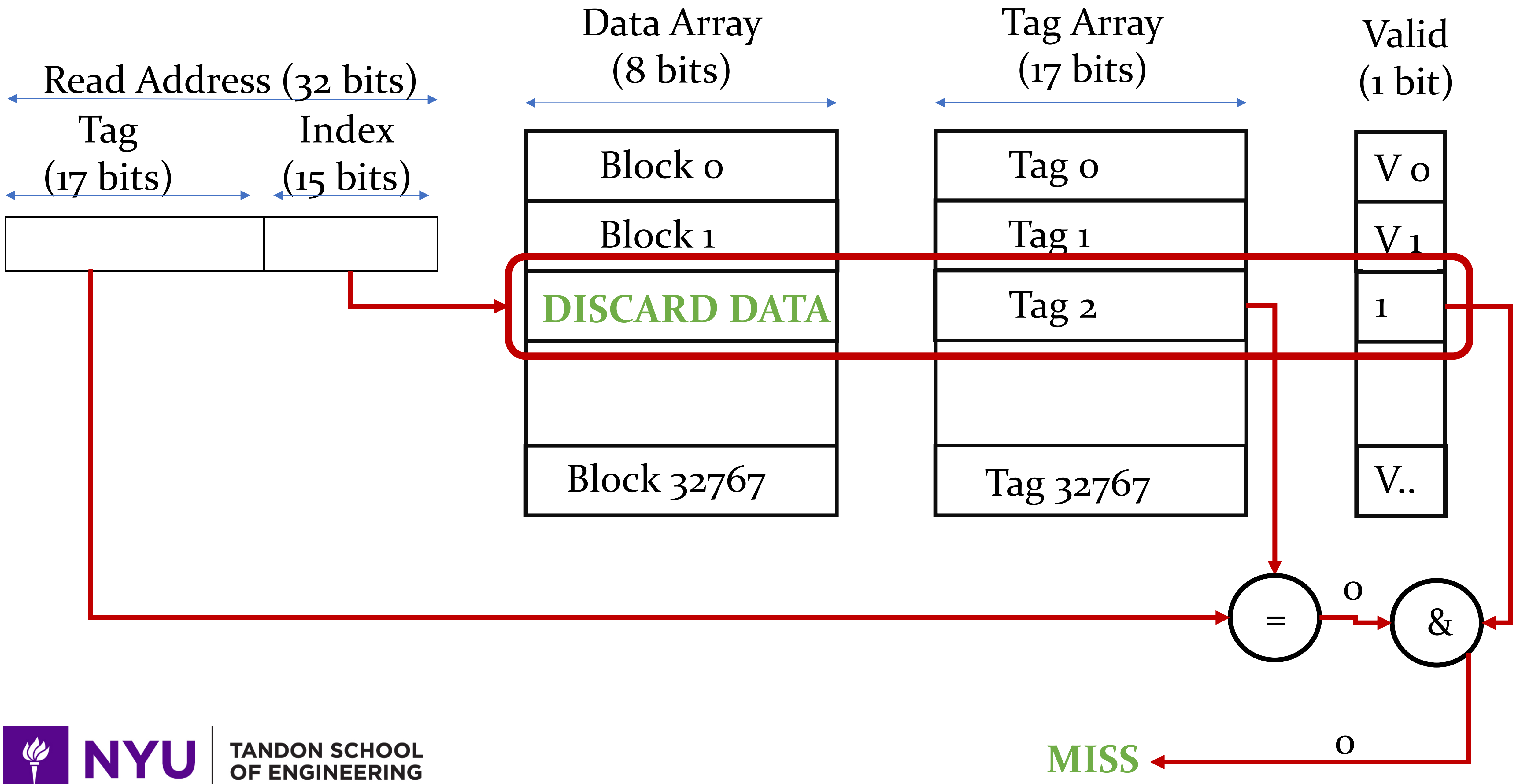
- Increased write bandwidth (multiple writes to same block)
- Potentially increased write latency
  (wait for write to propagate to lower levels?)

Write Through Cache (32 KB, Direct Mapped, 1 Byte Block)



NYU TANDON SCHOOL OF ENGINEERING

Write Through Cache (32 KB, Direct Mapped, 1 Byte Block)



NYU TANDON SCHOOL OF ENGINEERING

Write Miss Policies

What to do if a write access misses in the cache

- Write allocate policy
- Write no-allocate policy

Write-allocate Policy

- Treat like a read miss, allocate block in cache for write data
- Standard write hit actions follows
- Good match for write-back caches

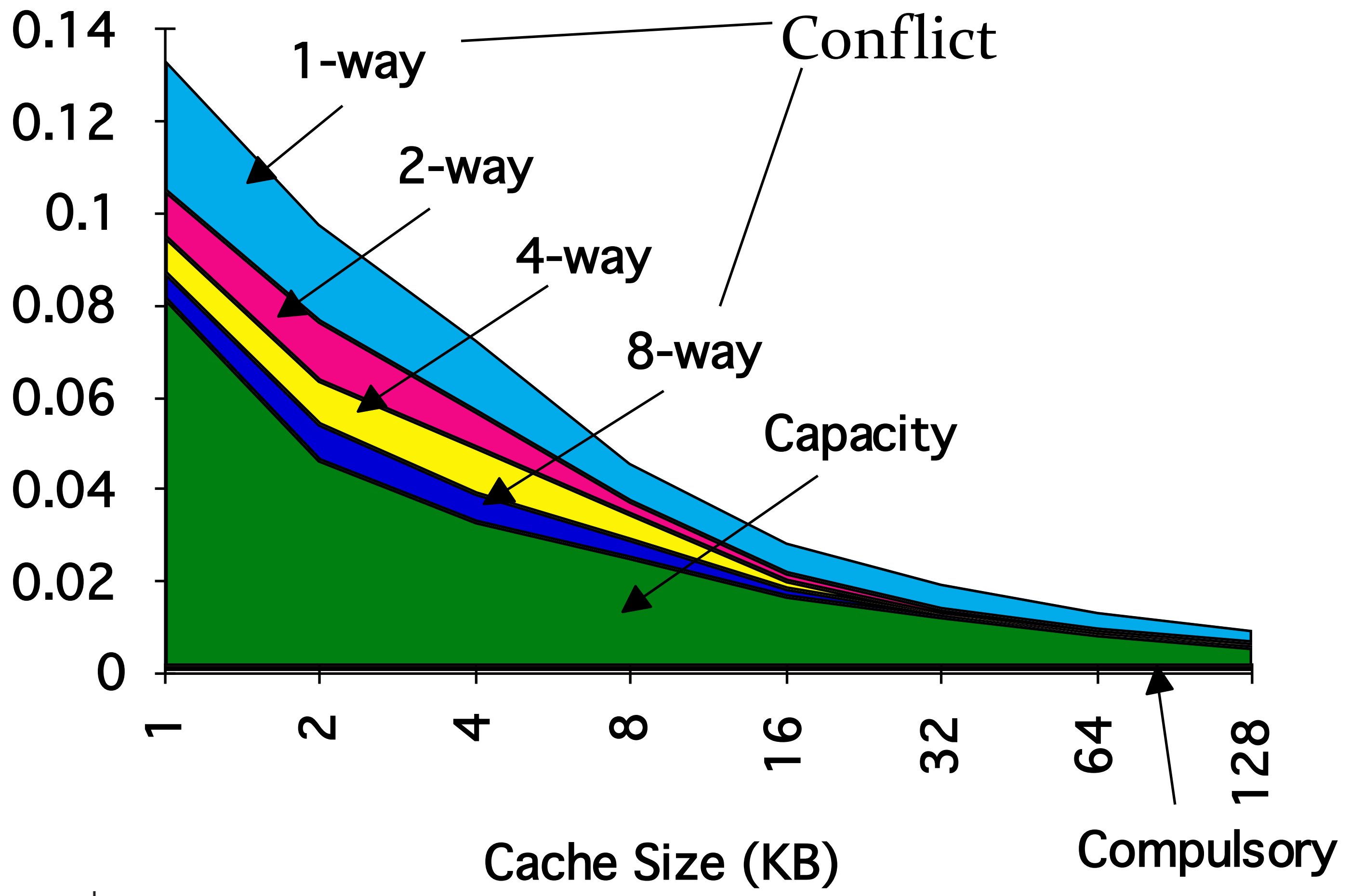
Write no allocate Policy

- Do not allocate a cache block for the write,
  instead forward write to the next level
- This implies that only a read access will result in allocations
- Goes well with write through policy

Reducing miss cost

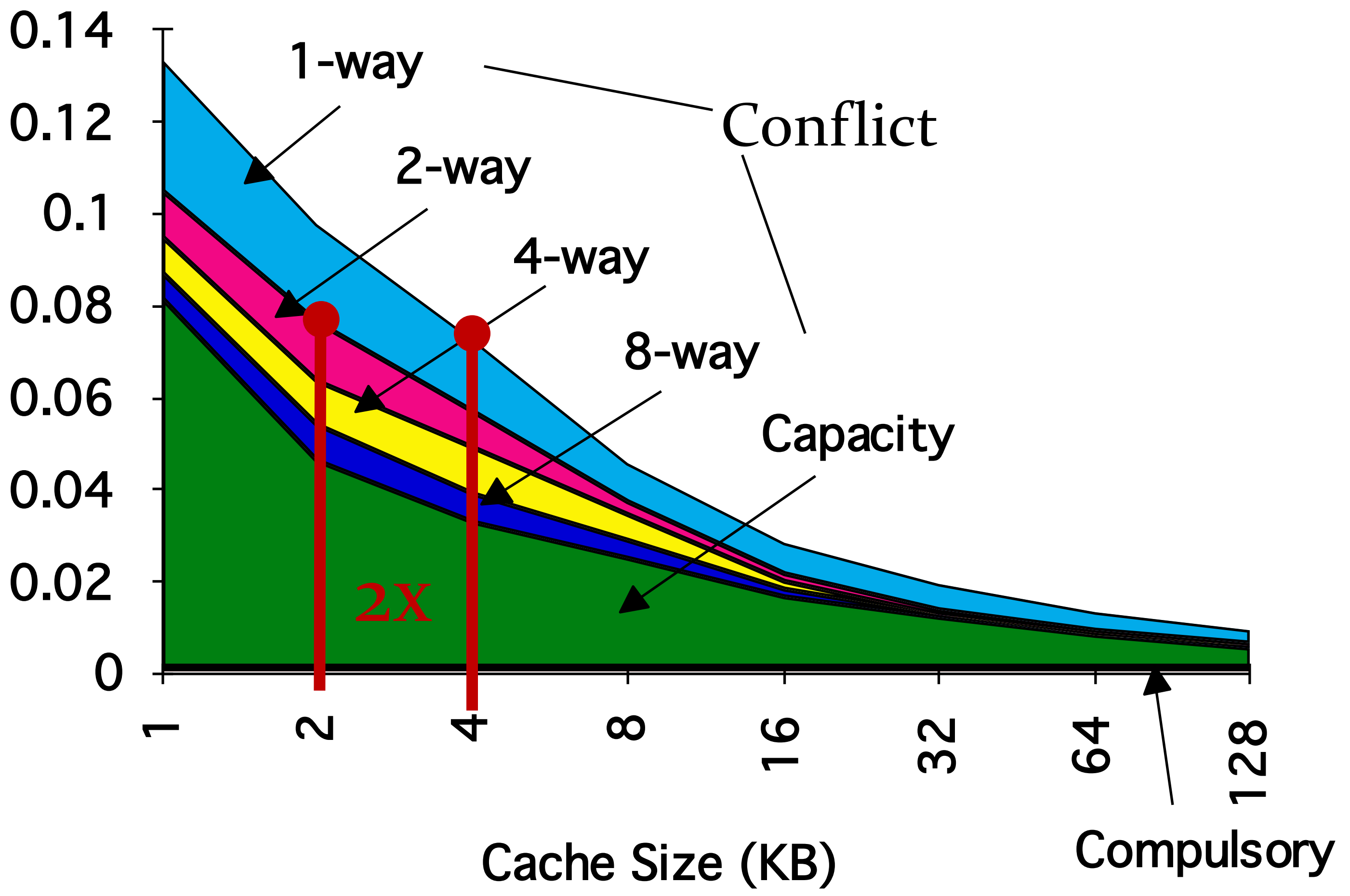
3Cs Absolute Miss Rate (SPEC92)

- Compulsory misses are a tiny fraction of the overall misses
- Capacity misses reduce with increasing sizes
- Conflict misses reduce with increasing associativity



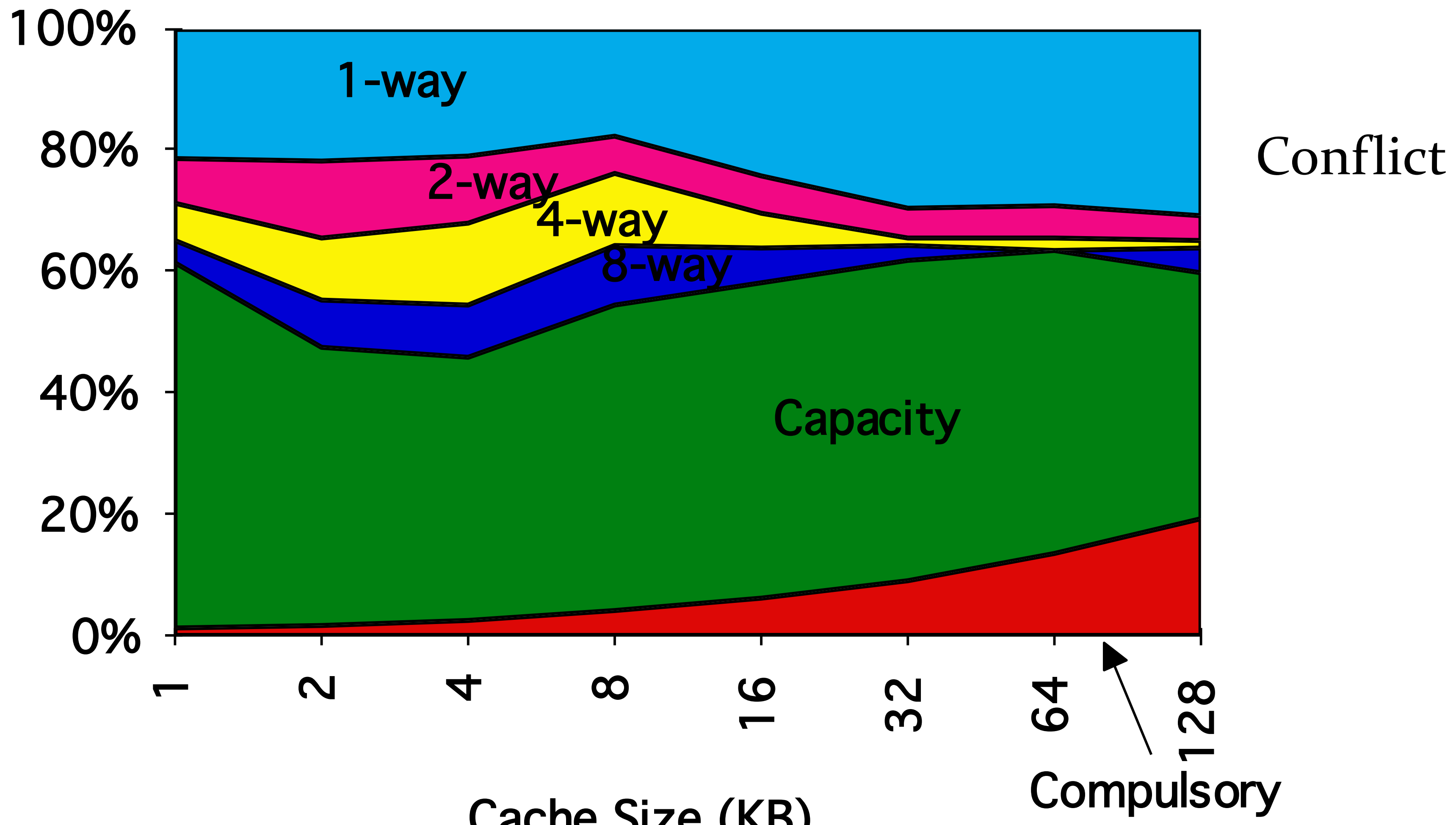
2:1 Cache Rule

Miss rate DM cache size X
~= Miss rate 2-way SA cache size X/2



NYU TANDON SCHOOL OF ENGINEERING

3Cs Relative Miss Rate



Reduce Miss Rate: Code Optimization

Misses occur if sequentially accessed array elements come from different cache lines

Code optimizations → No hardware change

- Rely on programmers or compilers

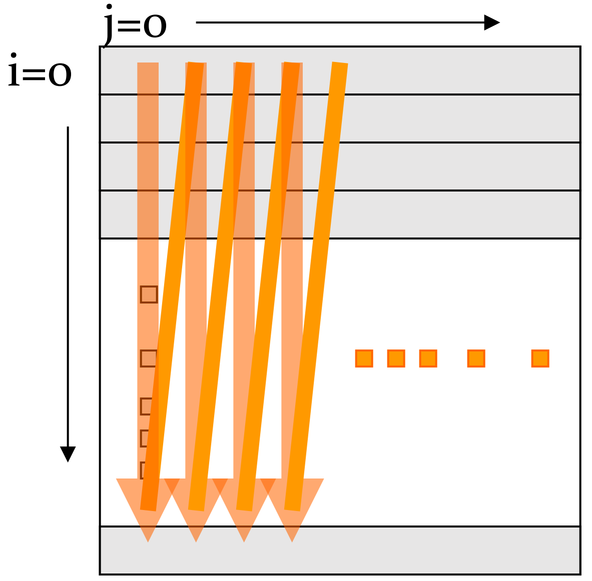
Examples:

- Loop interchange
  - In nested loops: outer loop becomes inner loop and vice versa
- Loop blocking
  - partition large array into smaller blocks, thus fitting the accessed array elements into cache size
  - enhances cache reuse

Loop Interchange

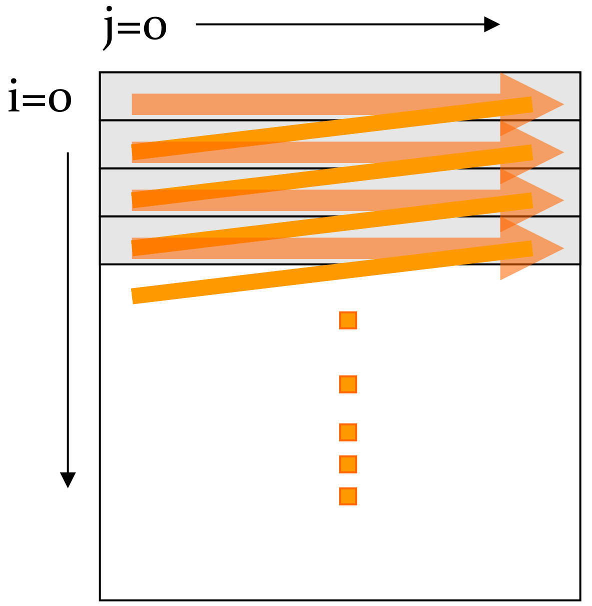
Row-major ordering

```
/* Before */
for (j=0; j<100; j++)
 for (i=0; i<5000; i++)
   x[i][j] = 2*x[i][j]
```



What is the worst that could happen?
Hint: DM cache

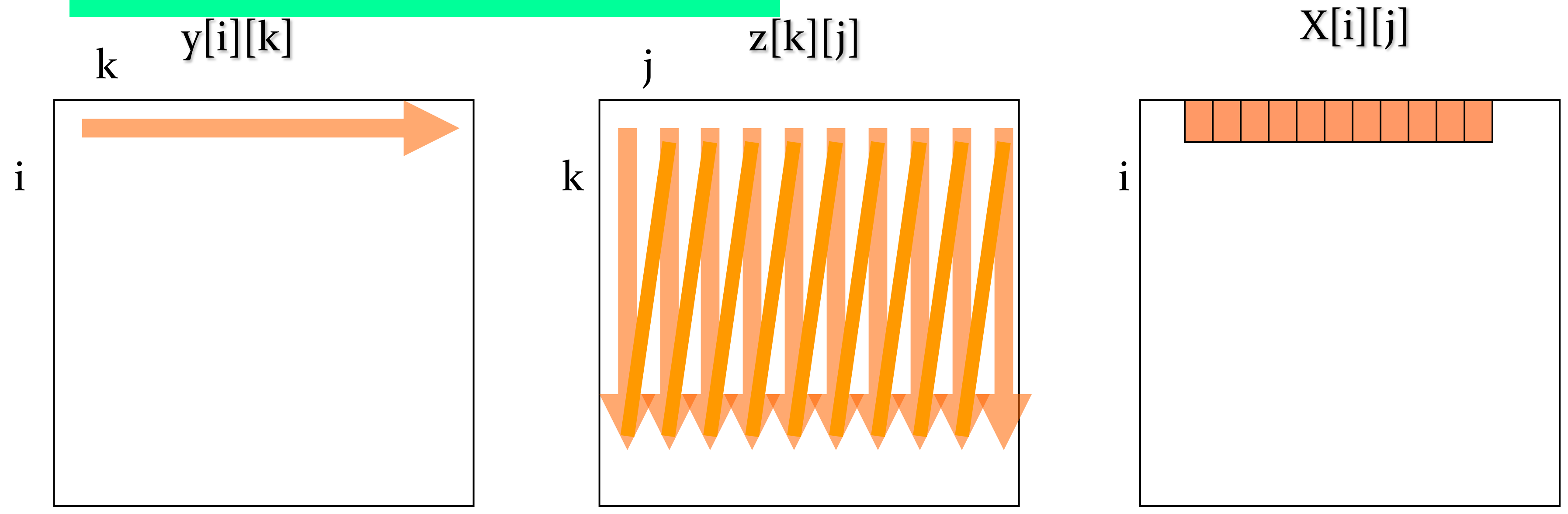
```
/* After */
for (i=0; i<5000; i++)
 for (j=0; j<100; j++)
   x[i][j] = 2*x[i][j]
```



Improved
cache efficiency

Loop Blocking

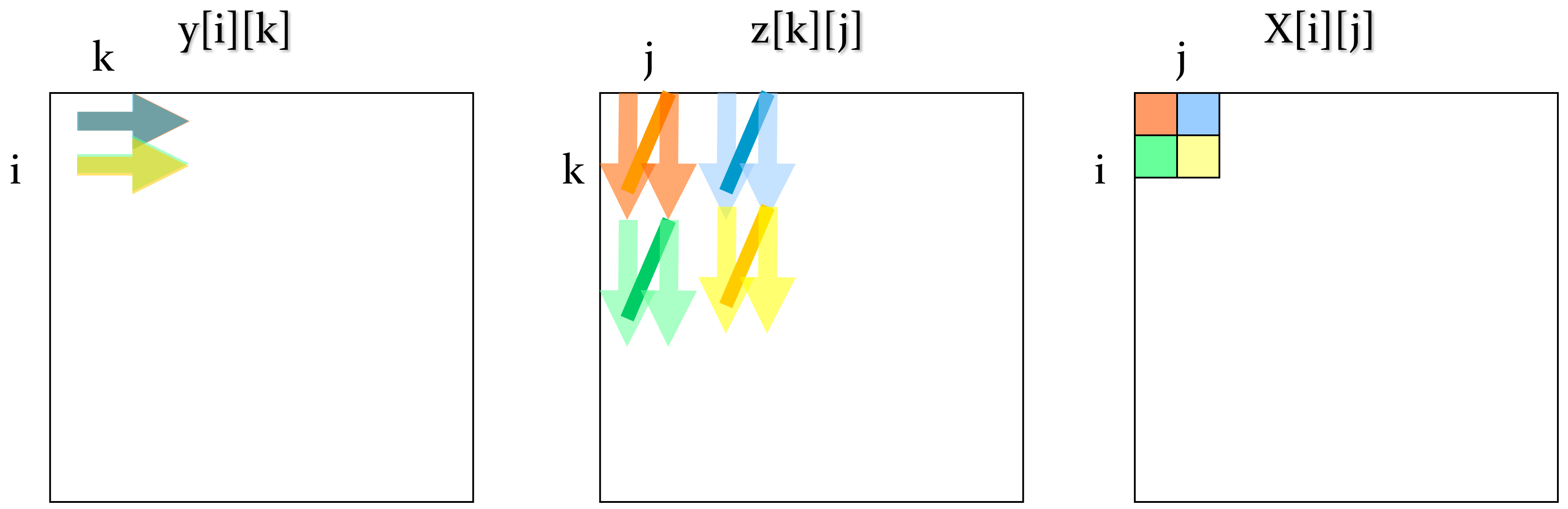
```
/* Before */
for (i=0; i<N; i++)
 for (j=0; j<N; j++) {
   r=0;
   for (k=0; k<N; k++)
      r += y[i][k]*z[k][j];
   x[i][j] = r;
 }
```



Does not exploit locality

Loop Blocking

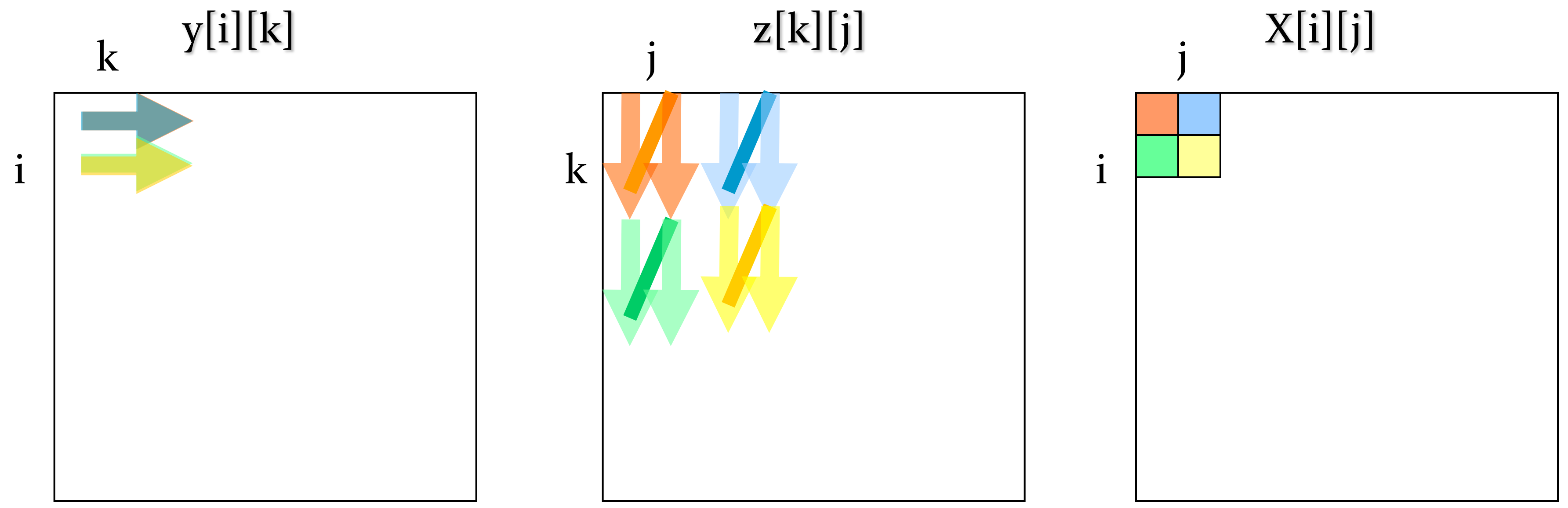
- Partition the loop's iteration space into many smaller chunks
- Ensure that the data stays in the cache until it is reused



See: "A Data Locality Optimizing Algorithm"
Monica Lam, PLDI '91

Loop Blocking

- Partition the loop's iteration space into many smaller chunks
- Ensure that the data stays in the cache until it is reused



This is super relevant.
This is tiling in DNNs (what TVM does).
If you can tile DNNs well, Google/NVIDIA needs you!

See: "A Data Locality Optimizing Algorithm"
Monica Lam, PLDI '91

Victim Cache

Jouppi [1990]:
4-entry victim cache removed
20% to 95% of conflicts for a
4 KB direct mapped data cache

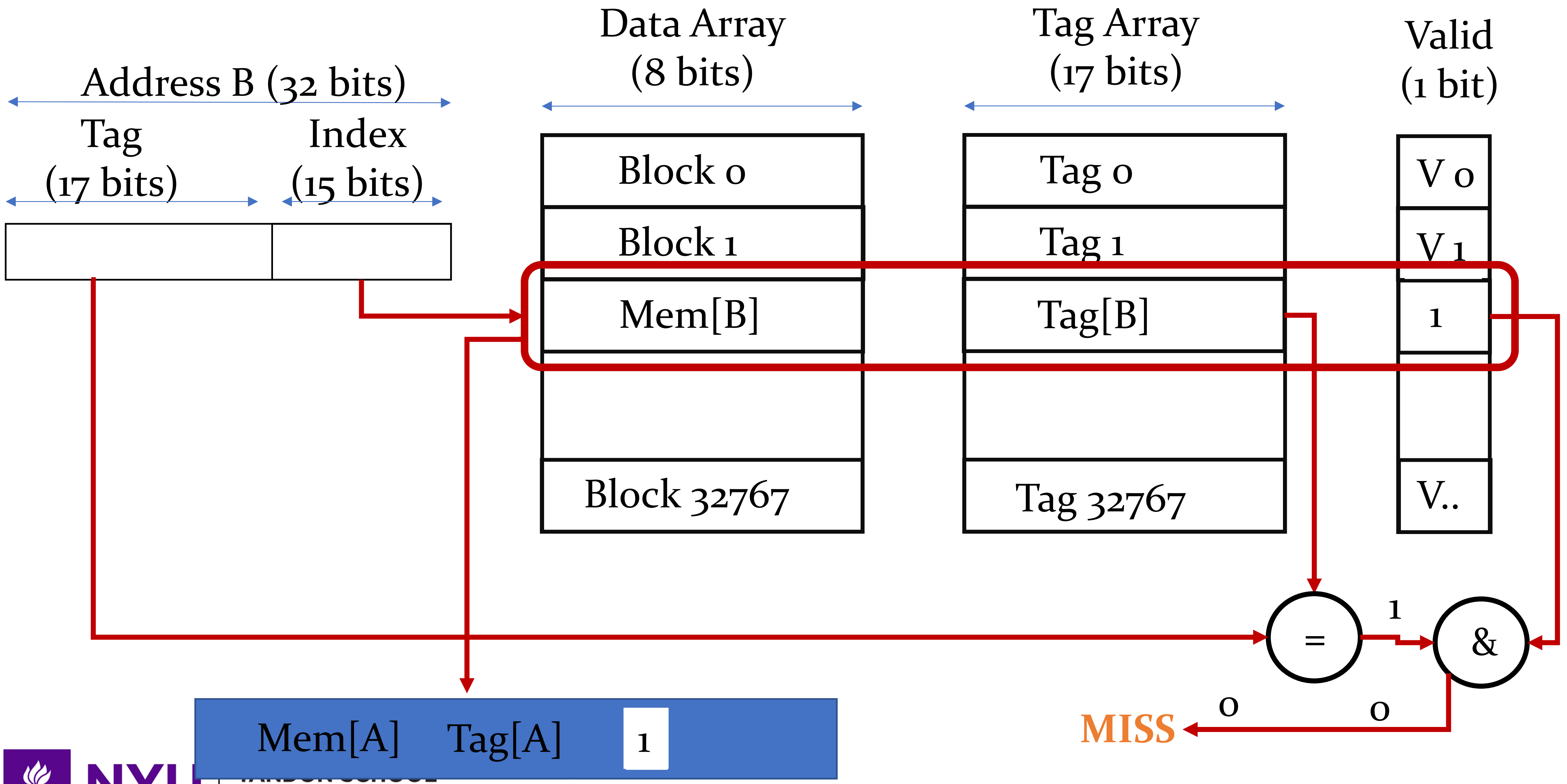
Direct mapped caches are “cheap”
but result in high conflict miss rate

A fully associative cache is expensive,
but has low conflict miss rate.. Can we get both?

A victim cache is a small (4-8 entry) fully associative cache that holds blocks evicted due to conflict misses

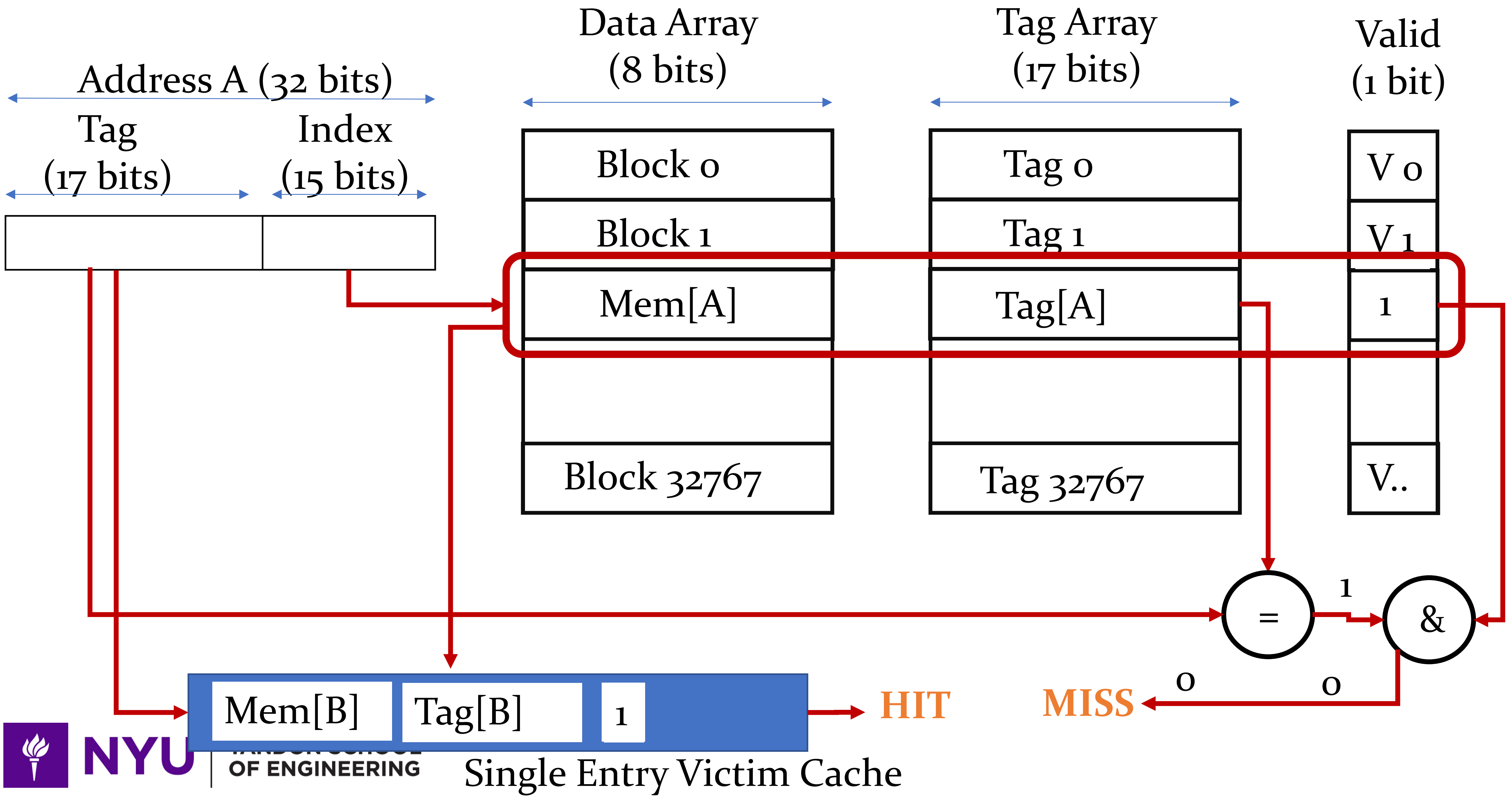
- On path between L1 and L2
- Checked on L1 miss
- Hit in victim cache -> swap with block in L1

32 KB Direct Mapped Cache with 1 Byte Blocks + Victim Cache

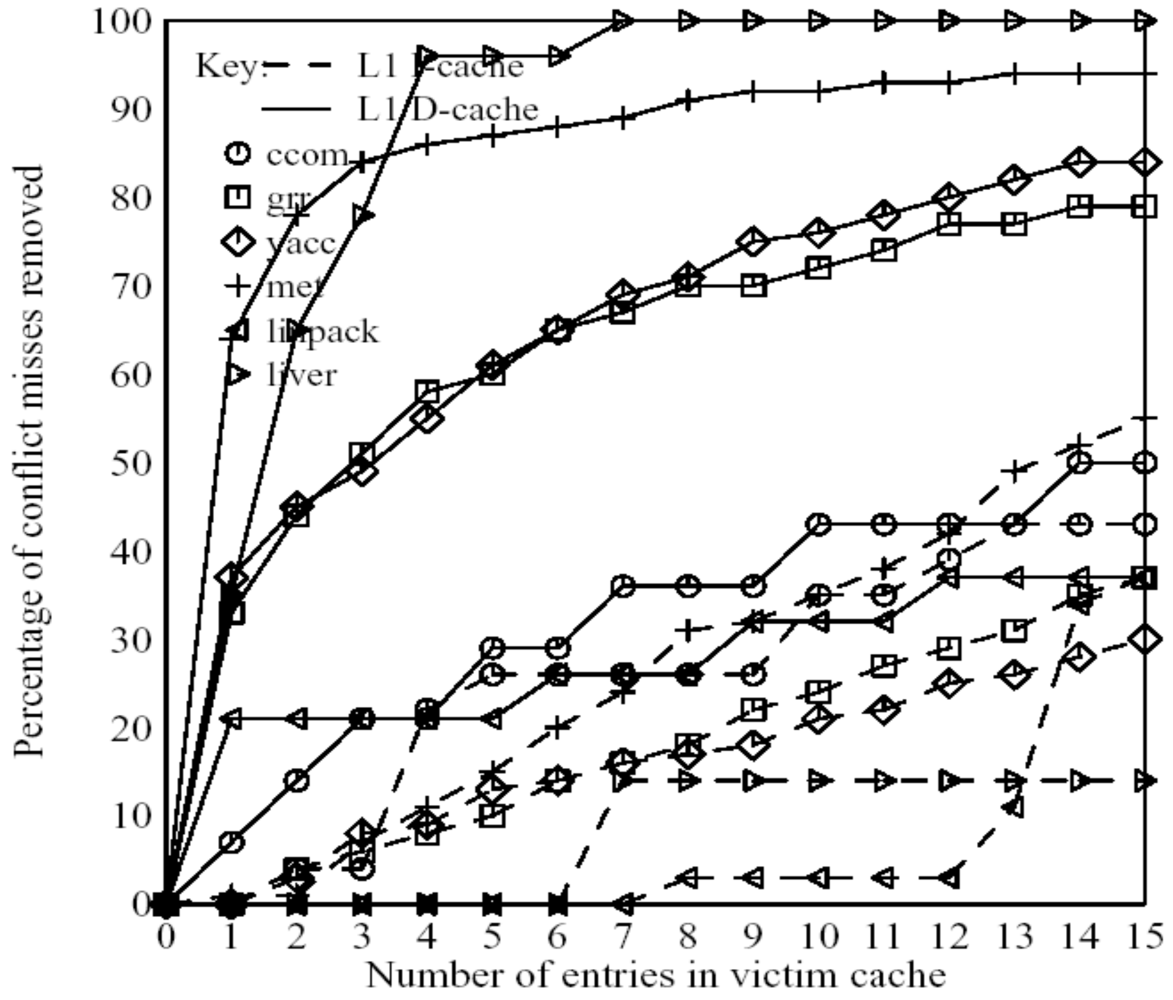


Single Entry Victim Cache

32 KB Direct Mapped Cache with 1 Byte Blocks + Victim Cache

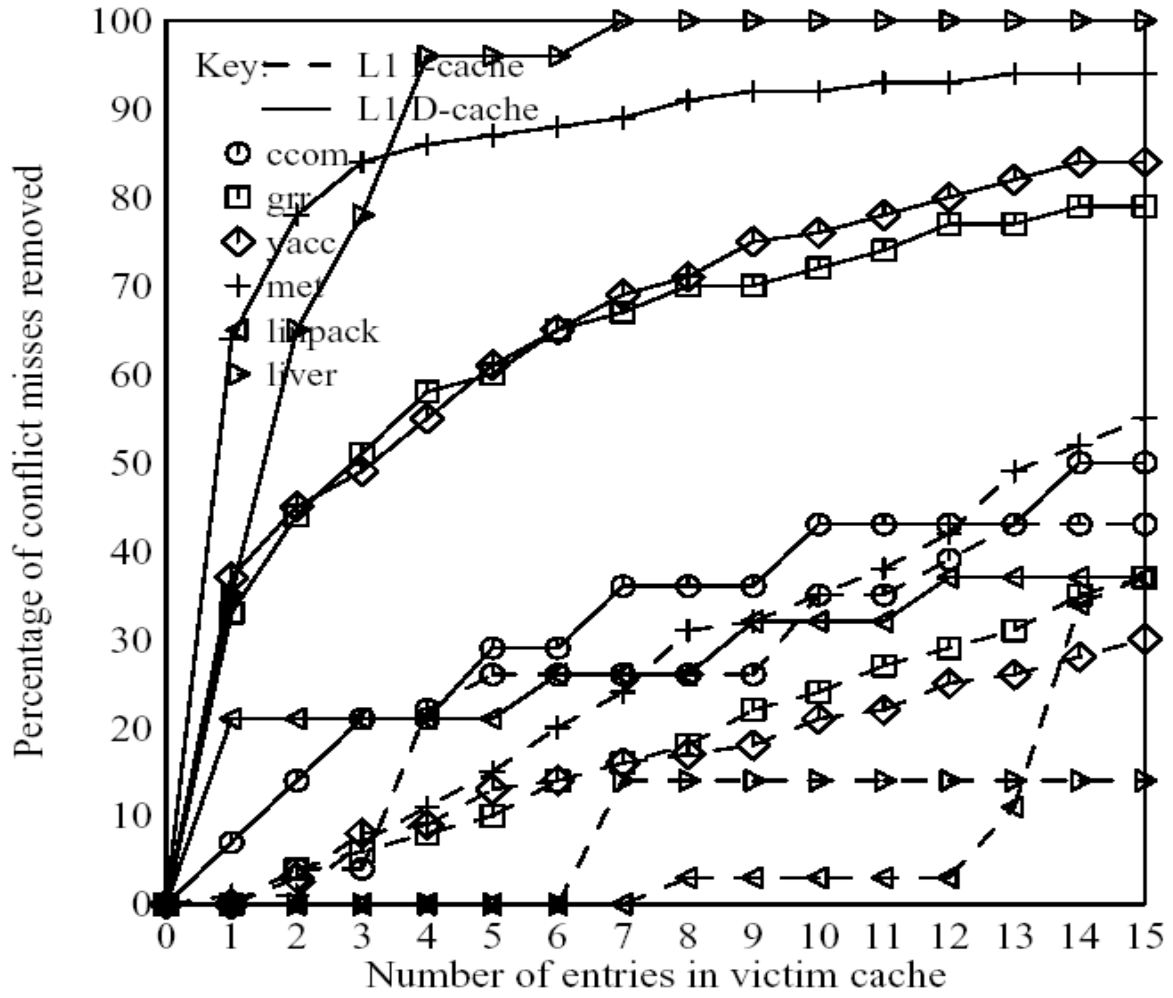


% of Conflict Misses Removed



Jouppi'90

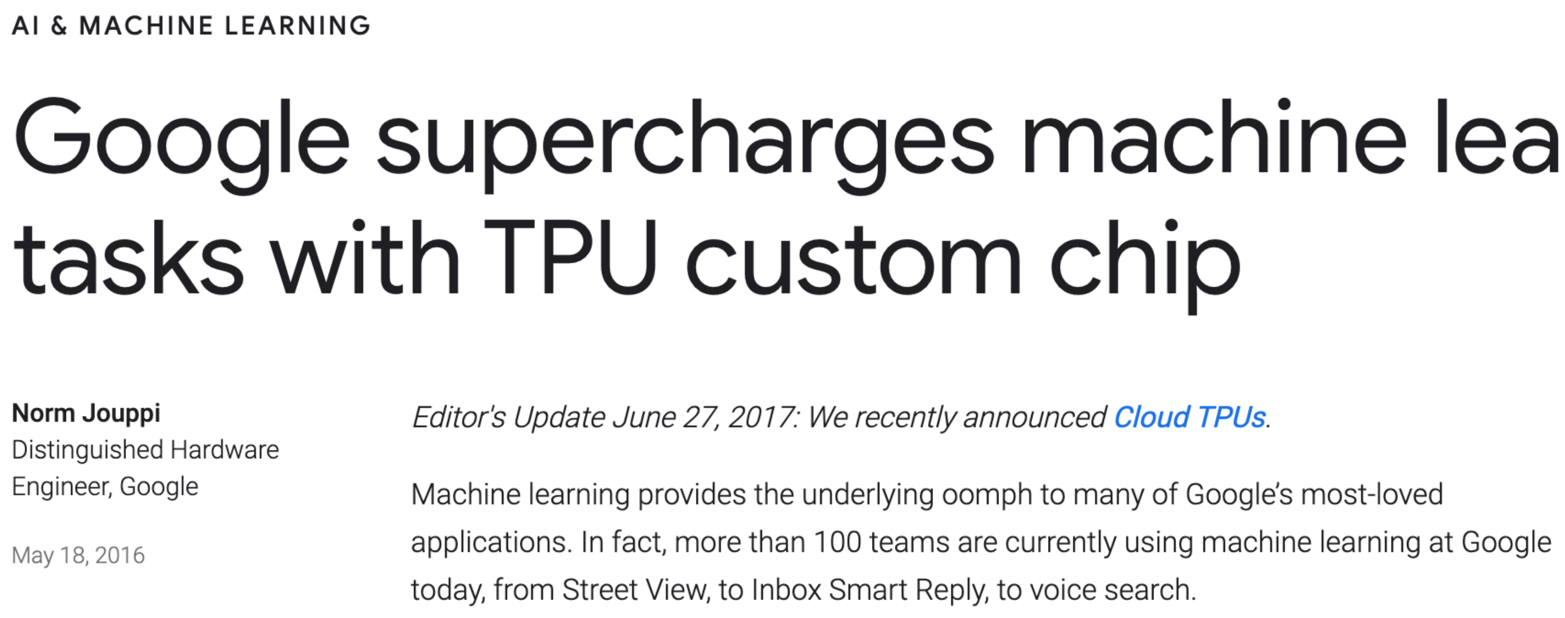
% of Conflict Misses Removed



Norm Jouppi…
What else did he do?

Jouppi'90

Built the TPU



AI & MACHINE LEARNING

Google supercharges machine learning tasks with TPU custom chip

Norm Jouppi
Distinguished Hardware Engineer, Google

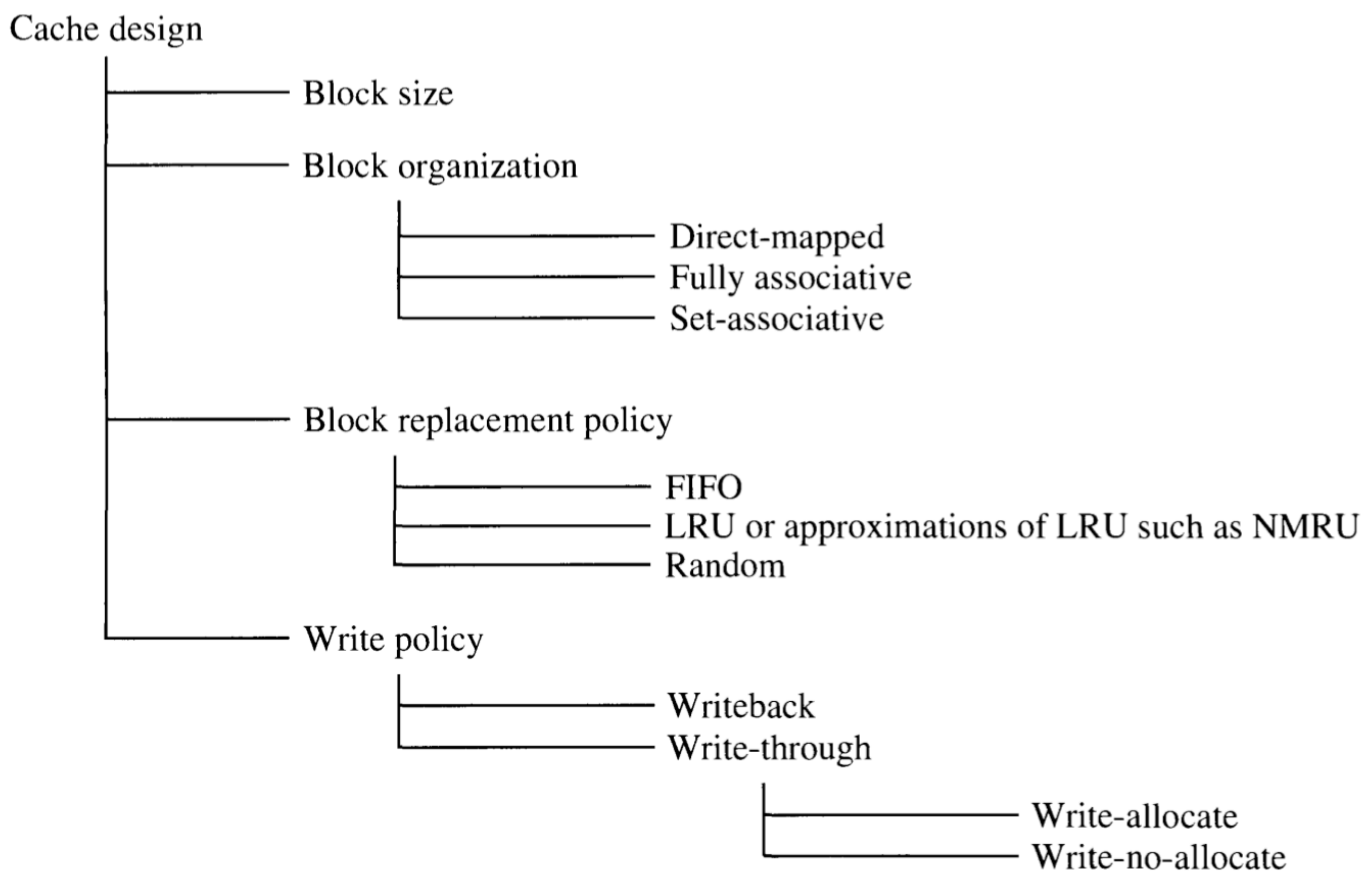
May 18, 2016

Editor's Update June 27, 2017: We recently announced ***Cloud TPUs****.*

Machine learning provides the underlying oomph to many of Google's most-loved applications. In fact, more than 100 teams are currently using machine learning at Google today, from Street View, to Inbox Smart Reply, to voice search.

NYU | TANDON SCHOOL OF ENGINEERING

Cache summary



NYU TANDON SCHOOL OF ENGINEERING

Virtual Memory

Virtual Memory

Use Main memory as a "cache" for disk

1) Give each program (process) its own view of memory
   a) When you write code, don't worry about others
   b) Gives larger memory view than may be available (4GB address space vs. 2GB DRAM)
2) Protection! Makes sure different programs can't access each others memory.

   Q: If you didn't have virtual memory, how would you have to program?

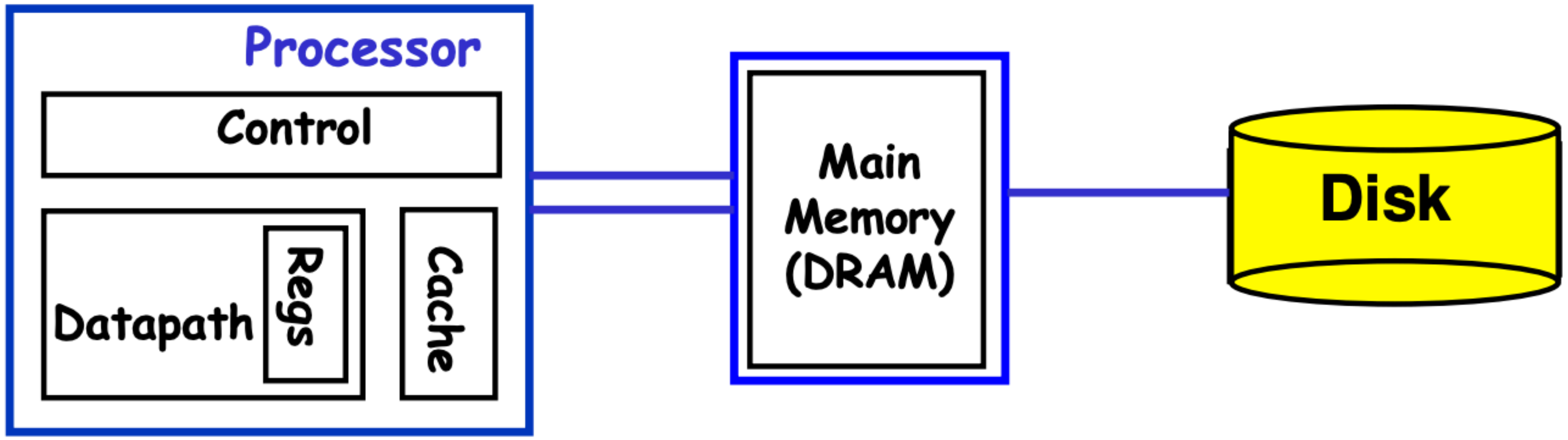
3) Key to VM is <u>translation:</u> Processor sees virtual addresses, memory sees physical addresses

VM – Cache effects

Not all program data needed at all times, DRAM is scarce resource

- VM allows main memory to be used effectively

Locality comes into play once again!



Virtual memory: Terminology

Page: A single unit of virtual memory. Like a line/block in caches

Frame: A single unit of physical memory (the page of PM)

Page fault: “Miss” in the virtual memory. No mapping exists between virtual and physical addresses, bring in from disk

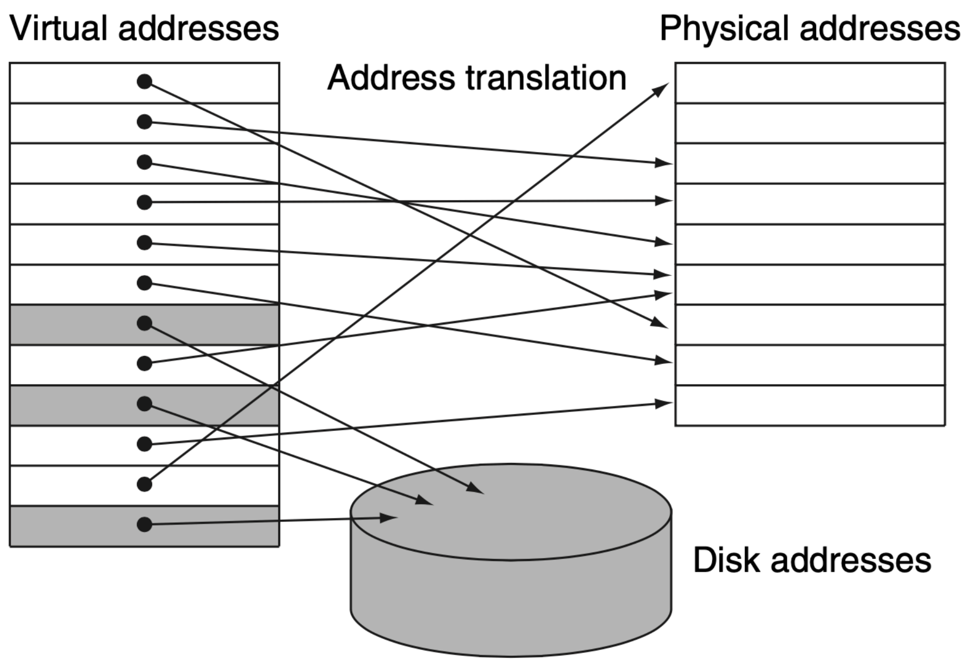
VM address: Address produced by the program/processor by the program

PM address: Address used to access main memory

Address translation: Process of mapping virtual addresses to physical ones

Relocation: Process of mapping physical addresses to virtual ones

Simple example



If a virtual address is not mapped,
then data exists on disk.
Unmapped accesses trigger page faults

Virtual memory: big picture

Each process has its own private "virtual address space" (4GB),

programs use "virtual addresses"

Each computer has a single "physical address space" (2GB),

sometimes called "real memory"

Address translation: mapping CPU's VM addr to Memory's PM addr

This allows some memory to be in disk, programmer doesn't know

Also allows multiple programs to use "real memory" at the same time

Addressing memory

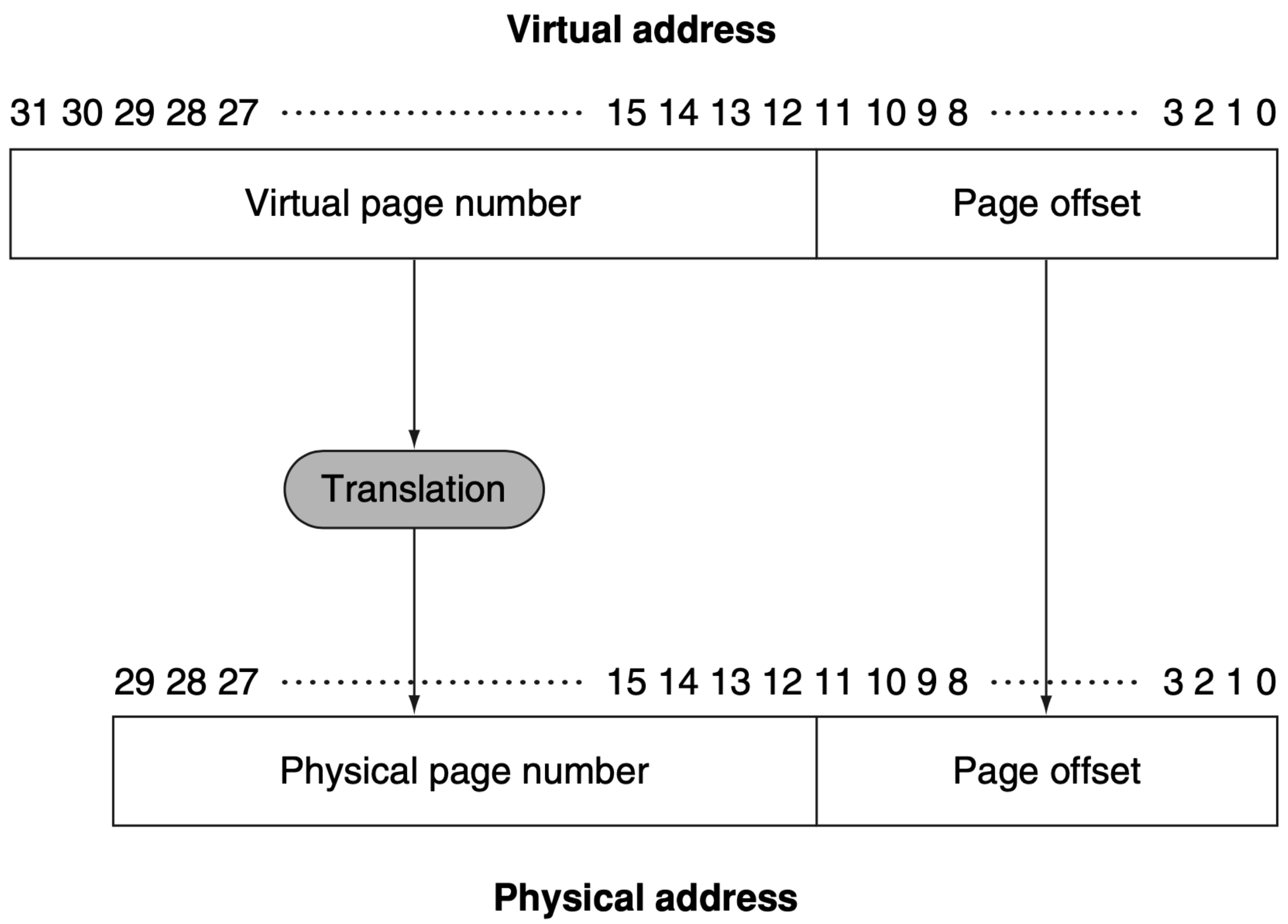
Virtual address has two parts: virtual page number and page offset

The size of a page is determined by the page offset

This means we don't have to translate it!

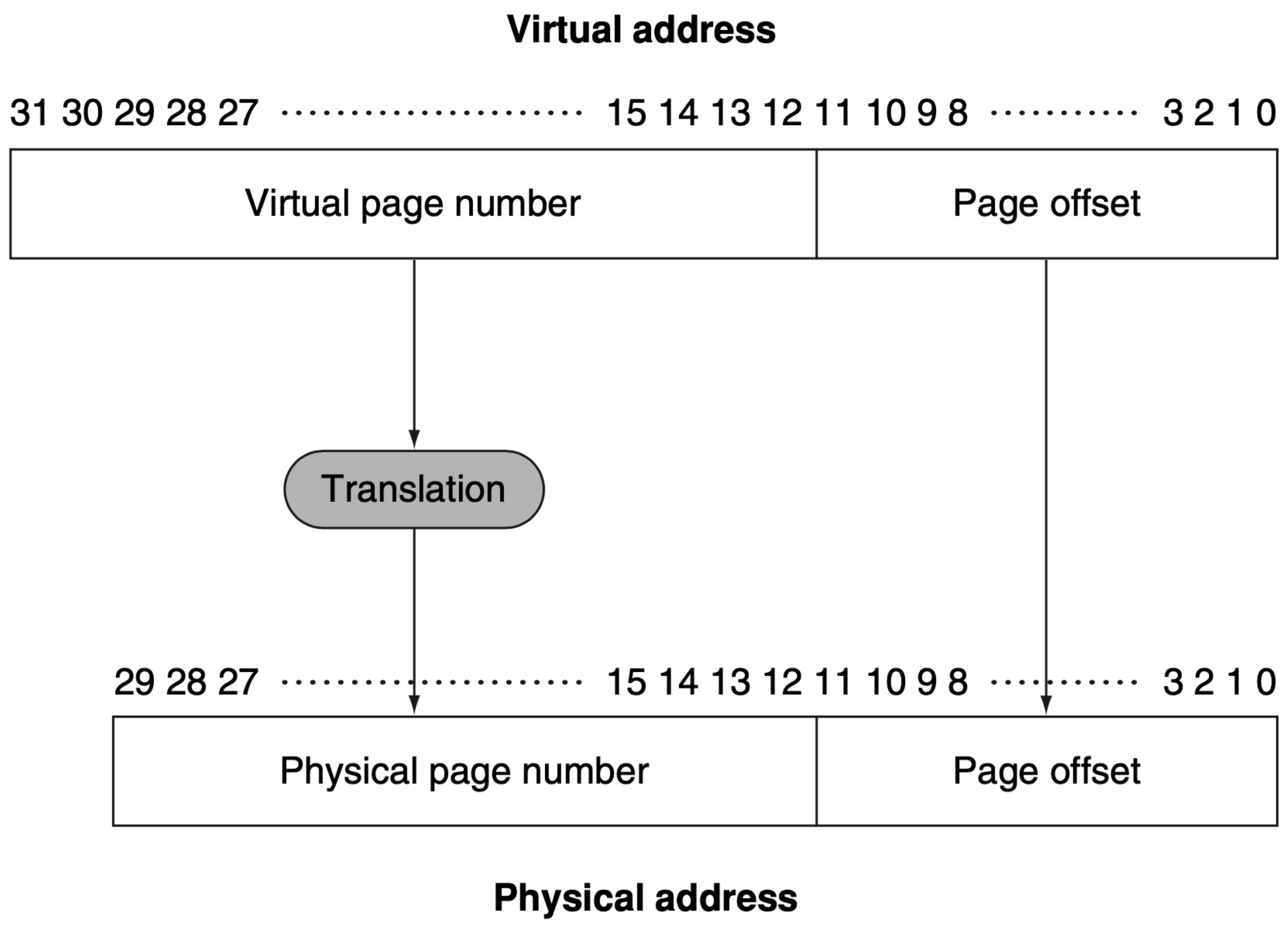
Translation: Convert a program's virtual page number to the machine's physical page (frame) number

Address translation - simplified



Quiz:
How large are pages?
How large is the "real memory"?

Address translation - simplified



Quiz:
How large are pages?
How large is the "real memory"?
How large is the virtual memory?

Pages = 2^12 = 4KiB
Real mem = 2^30 = 1GB
Virtual mem = 2^32 = 4GB

Virtual memory design choices

Key objective: Avoid page faults! Page faults can take millions of cycles to resolve.

Most of this time is spent getting the first word

- Amortize costs once we're serving a fault using large pages
- Today: 4-16KiB common. 32-64KiB in servers (and beyond!)

Organize real memory to reduce faults. Which placement scheme?

- Fully associative

Who should handle page faults?

- SW: time in disk access (think about Amdahl's law)
- Complex replacement algorithms OK because faults take so long

What about writes, write through or write back?

- Write back! Want to minimize bandwidth

Placing and finding pages

Placement:

OS is allowed to map any virtual page to any physical page

Using full associativity minimizes conflicts for limited resource

What’s the problem with fully-associative structures?

How do we find the data? Takes long time in caches,

or very expensive parallel access

Page Table: a special, auxiliary data structure to tack VA -> PA mappings

Page tables

A page table takes virtual addresses as input and outputs physical address

- Track all virtual to physical mappings
- This means they can get quite large

Page tables live in memory

- A special register (page table register) points to the start of
  - each processes page table

Quiz: How large is a single program's page table given:

- 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry

Page tables

A page table takes virtual addresses as input and outputs physical address

- Track all virtual to physical mappings
- This means they can get quite large

Page tables live in memory

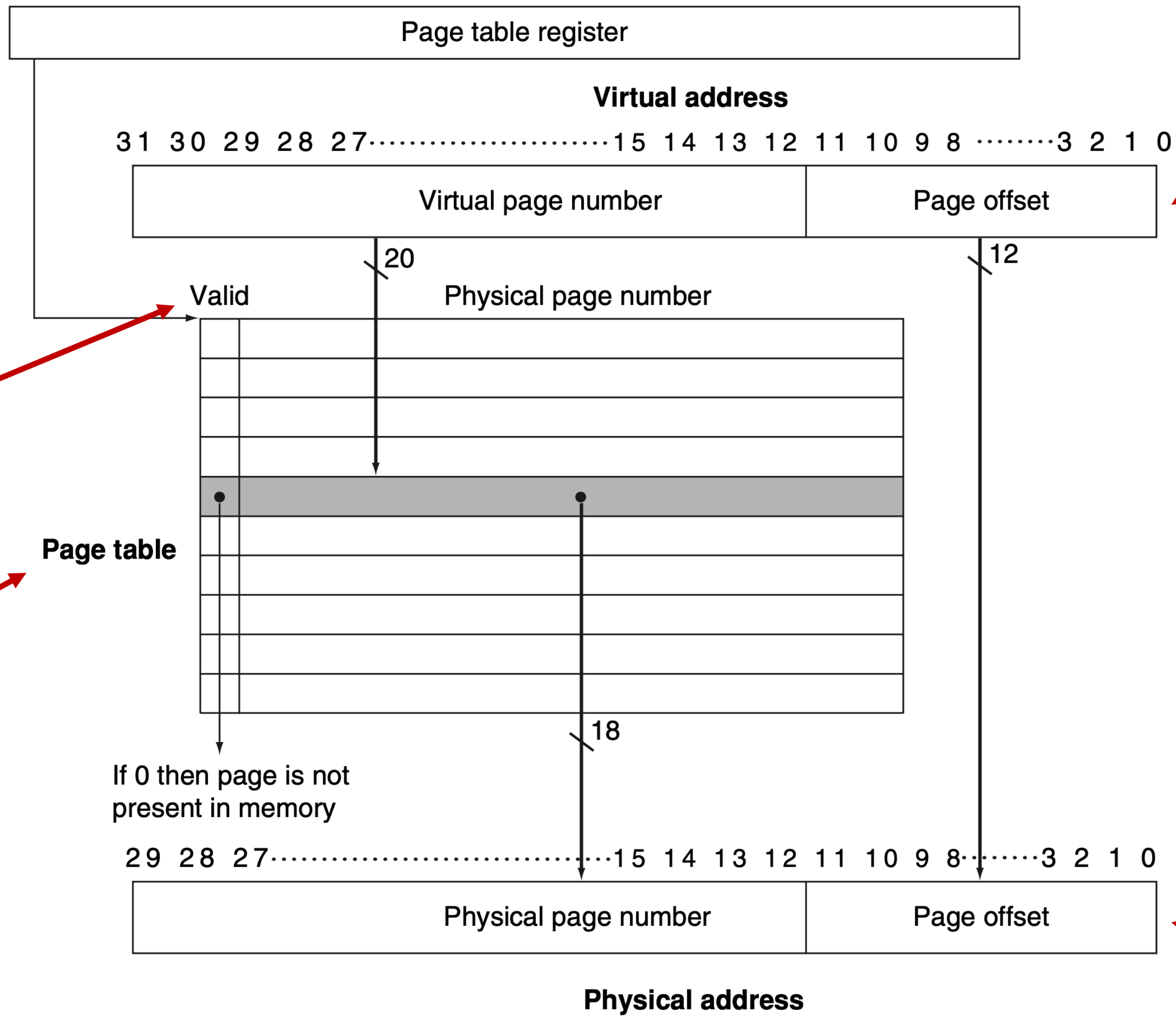
- A special register (page table register) points to the start of
  - each processes page table

Quiz: How large is a single program's page table given:

- 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry
- Num entries: 2^32 / 2^12 = 2^20.
- Total size: 2^20 * 2^2 <bytes per PTE> = 4 MiB

NYU | TANDON SCHOOL OF ENGINEERING

Page table example



Base address

Program Address (VA)

Valid bit: indicates whether the mapping exists or not

Also lives in memory..

Real address (PA)

Page faults

If we access a PTE and the valid bit is zero we incur a page fault

Raise an exception and give control to the OS

OS will go out and find the data stored on disk, bring it into main memory, and create a valid mapping in the page table

When process starts, OS creates “swap space” on disk

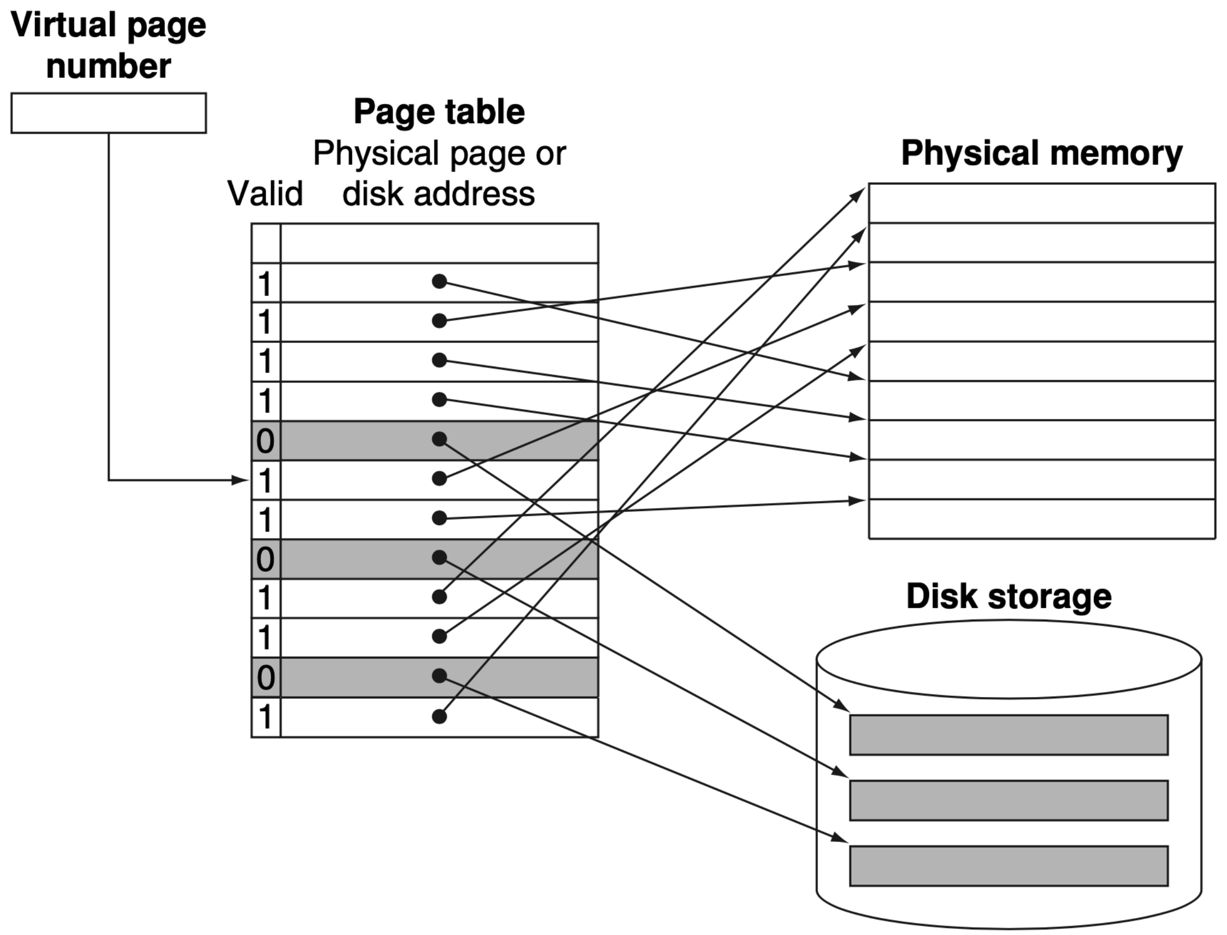
This is a pre-allocation of all virtual addresses

OS creates data structure for where all program’s VM addresses are stored on disk

This can be part of the page table or separate structure

Usually separate structures, enables PT optimizations

Page faults



Page faults are bad, because then we have to go to disk.

But the page table lives in main memory..

If we assume caches work using physical addresses, what does this mean?

How can we fix this?

Translation Lookaside Buffer (TLB)

Each time we need to access memory, actually makes 2 trips!

One for translation, one for access event

We can cache virtual to physical address translations

This is what the TLB does:

Leverage locality to store recently used translations

TLB is why professors spend so long thinking about paper names and titles

Terrible name. Translation Cache much better.

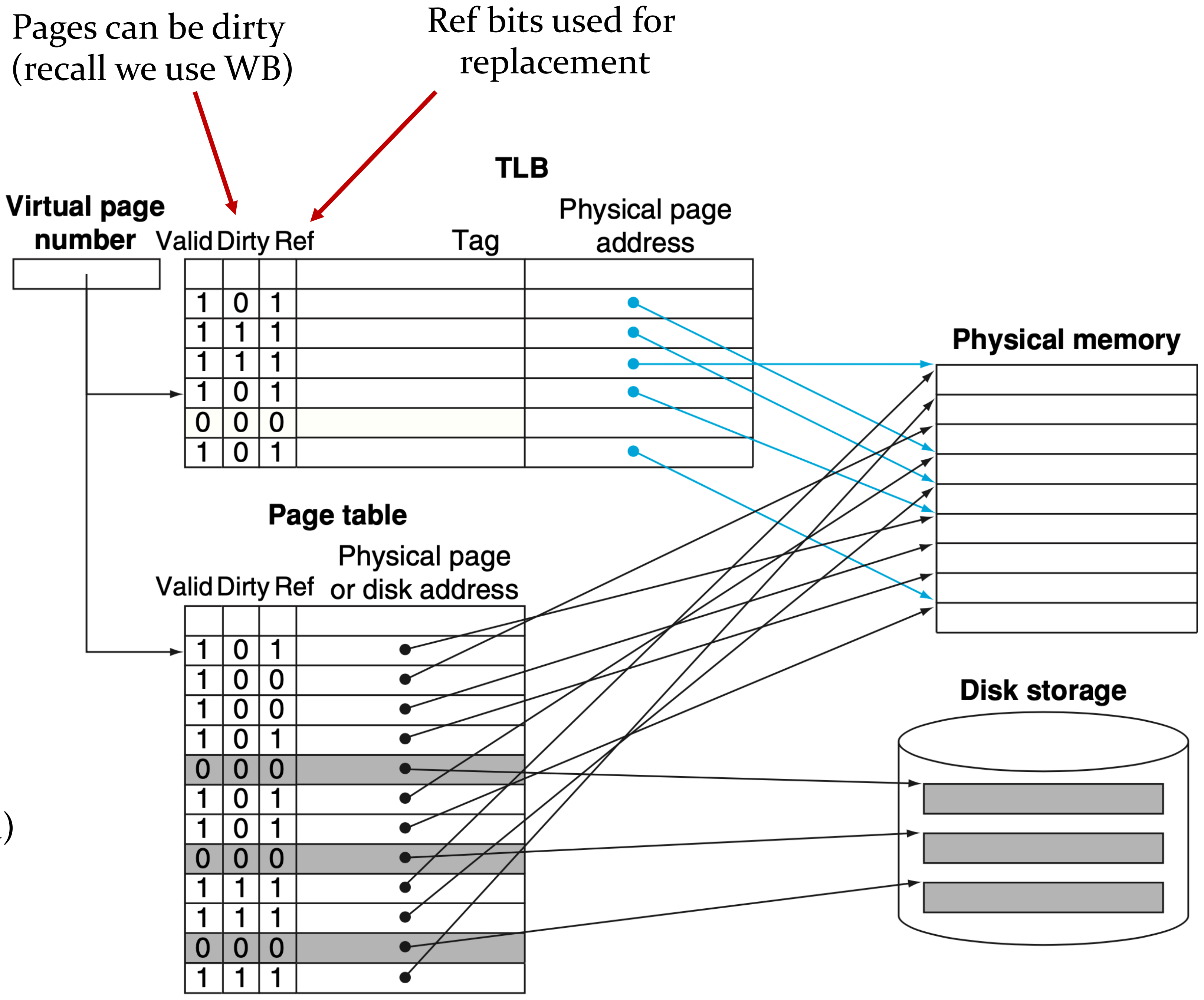
TLB example

TLB holds a subset of all VA->PA
translations

Hits in the TLB avoid going to the PT
Avoid 1 memory access!

Use high associativity,
we really don't want to go memory..

Note:
TLB is a cache (tag and data)
Page Table is not a cache (just data)



NYU TANDON SCHOOL OF ENGINEERING

TLB access process

On every memory event, access the TLB:

If TLB hit: the physical page number is used as the PA,

ref bit turned on. If write, turn on dirty bit

If TLB miss: Determine if simple TLB miss of page fault

If PTE valid, processor looks up VA in PT,
loads value to TLB, replays access

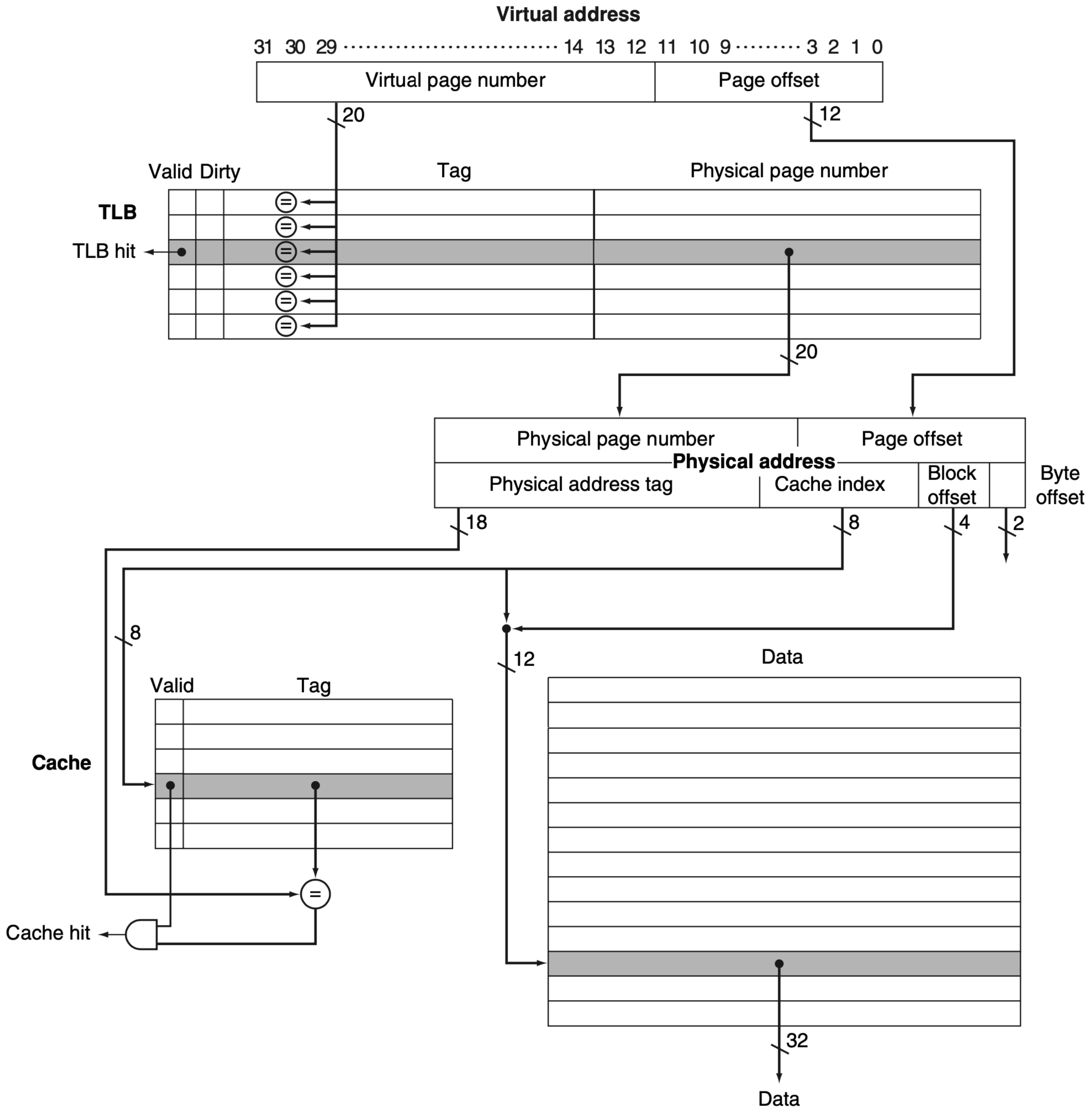
If page fault, invoke OS, trigger page fault event, go to disk,

create mapping in PT, load mapping into TLB, replay access

On eviction, only have to write back reference and dirty bit
Mappings cannot change in the TLB!

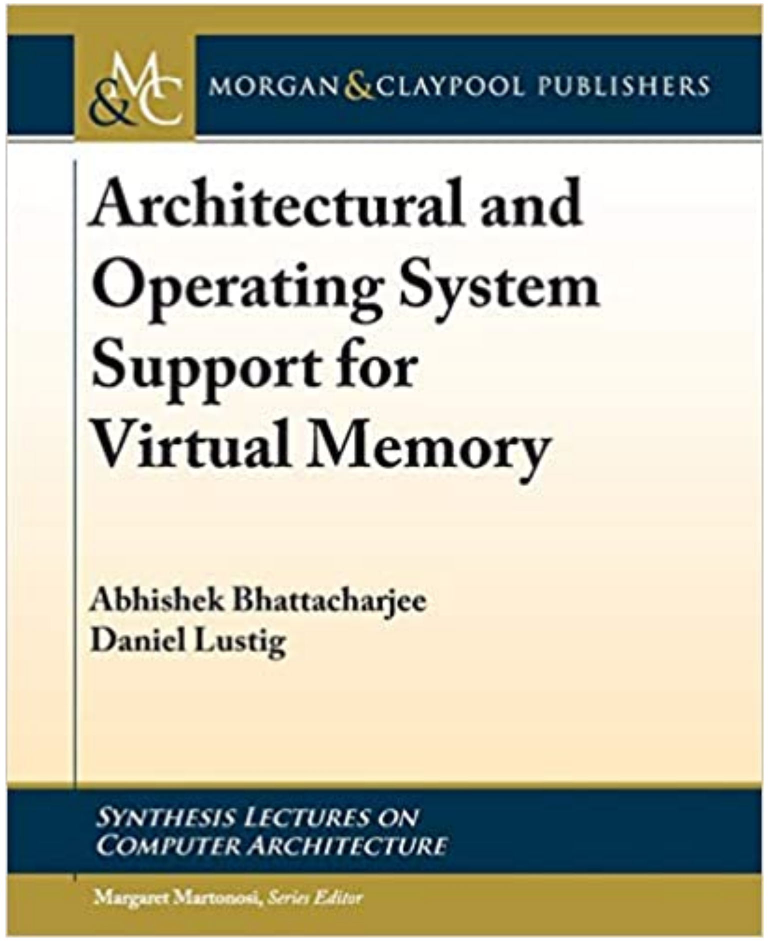
Whole picture

Wait.. What's going on with this cache??
The data has more entries than tag array



NYU TANDON SCHOOL OF ENGINEERING

If you want to know more check out



MORGAN & CLAYPOOL PUBLISHERS

Architectural and Operating System Support for Virtual Memory

Abhishek Bhattacharjee
Daniel Lustig

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, Series Editor

Next time

Tomasulo's algorithm for out-of-order processing