Caches

Computer Architecture
ECE 6913

Brandon Reagen

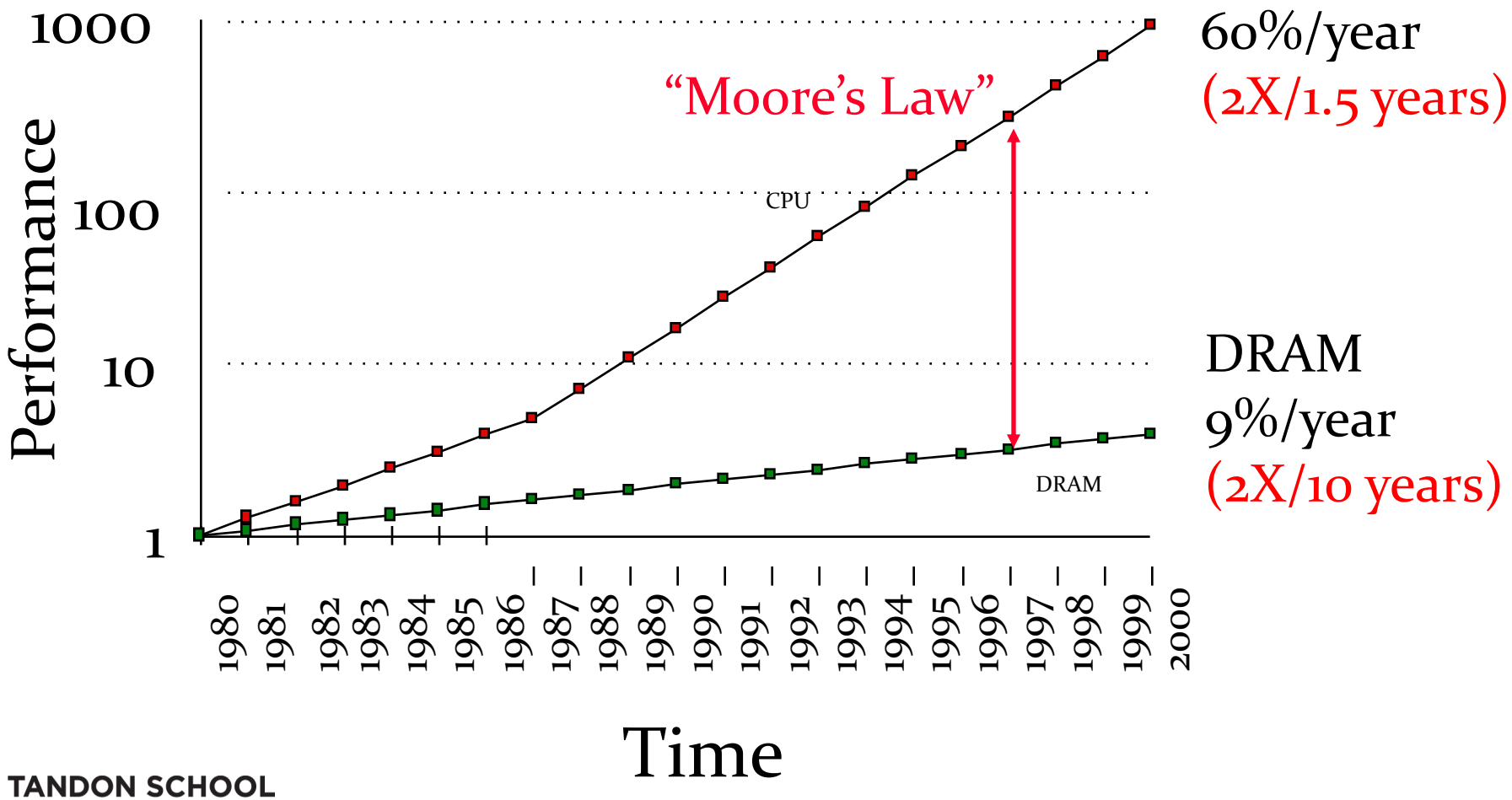
NYU TANDON SCHOOL OF ENGINEERING

Announcements

1) Lab1 grades released
    1) If you didn't submit contact me and the TAs immediately
    2) If you struggled, please meet with the Tas
2) Lab2
    1) How's it going?
    2) Please check in with the TAs even if things are going well
3) HW solutions will be posted soon
4) Practice mid-term upcoming
5) Reminder: Mid-term next week!
    (No class, will have Zoom session)
6) Today: More caches!

Why Care About Memory Hierarchy?

Processor-DRAM Performance Gap grows 50% / year

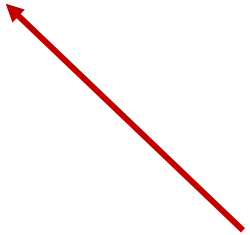


What can we do about it?

Tennent of architecture: “Smaller is faster.”

How would you solve this?
“Temporary” buffers between the register file and memory.

microArchitecture or Architecture technique?



Why is this so important?
It’s invisible to the programmer!

Caches invented in 1965 by Maurice Wilks

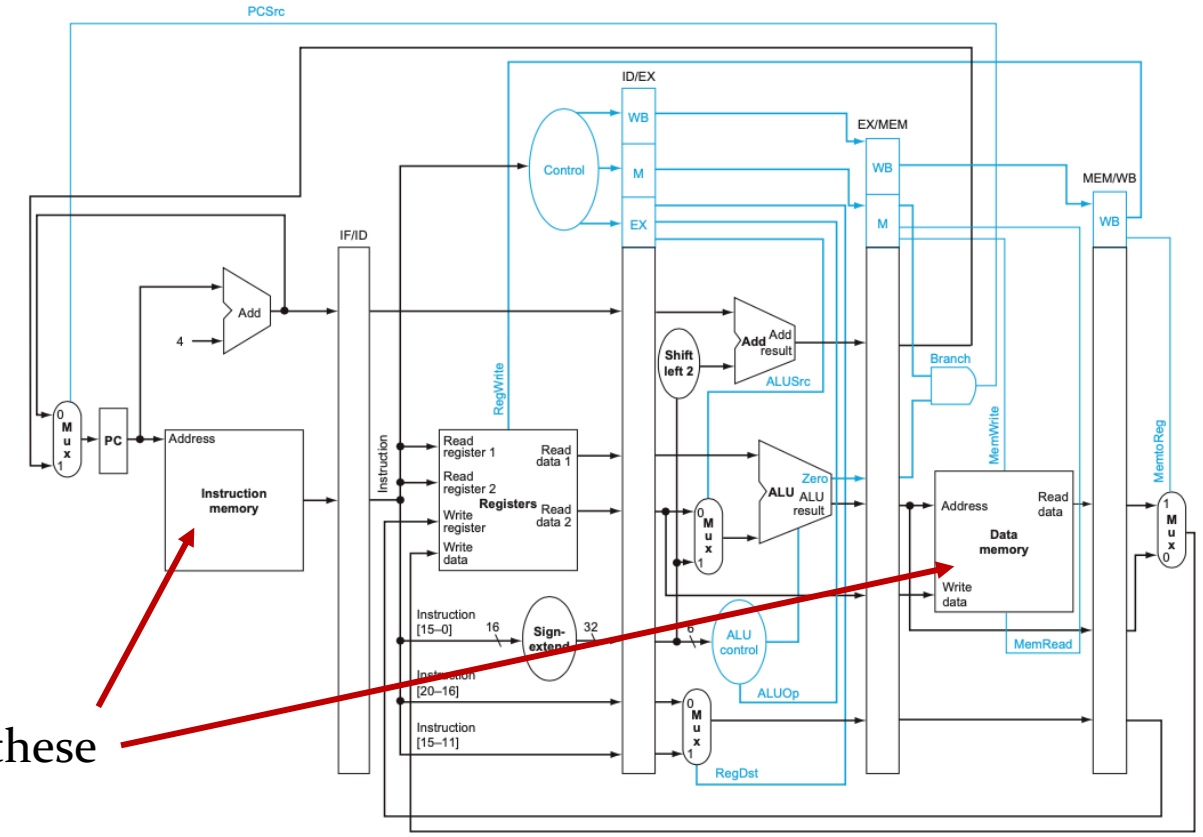
Cache paper was 1.5 pages long.
(Today's papers are 12 pages.)

Also build EDSAC
And invented microprogramming.

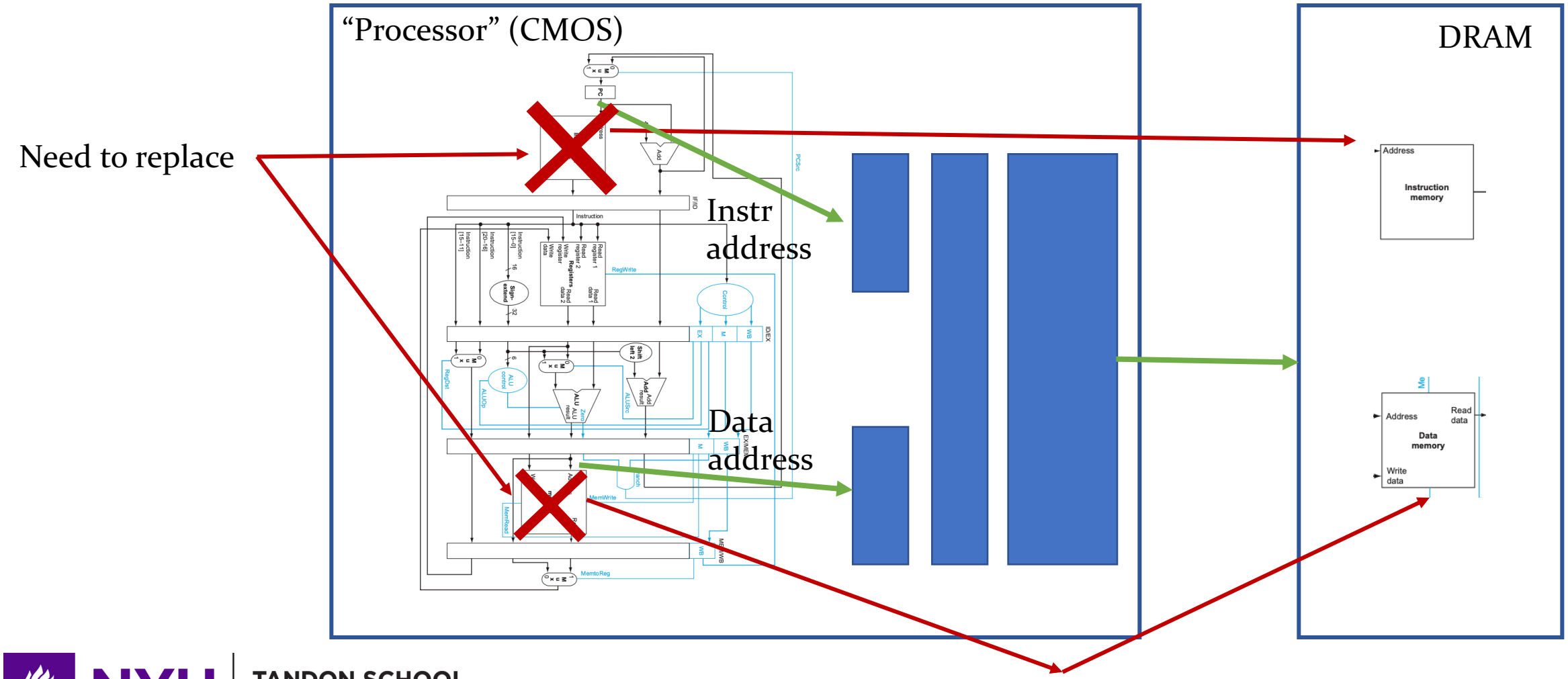
Pretty productive guy.

Maurice Wilkes in 1980

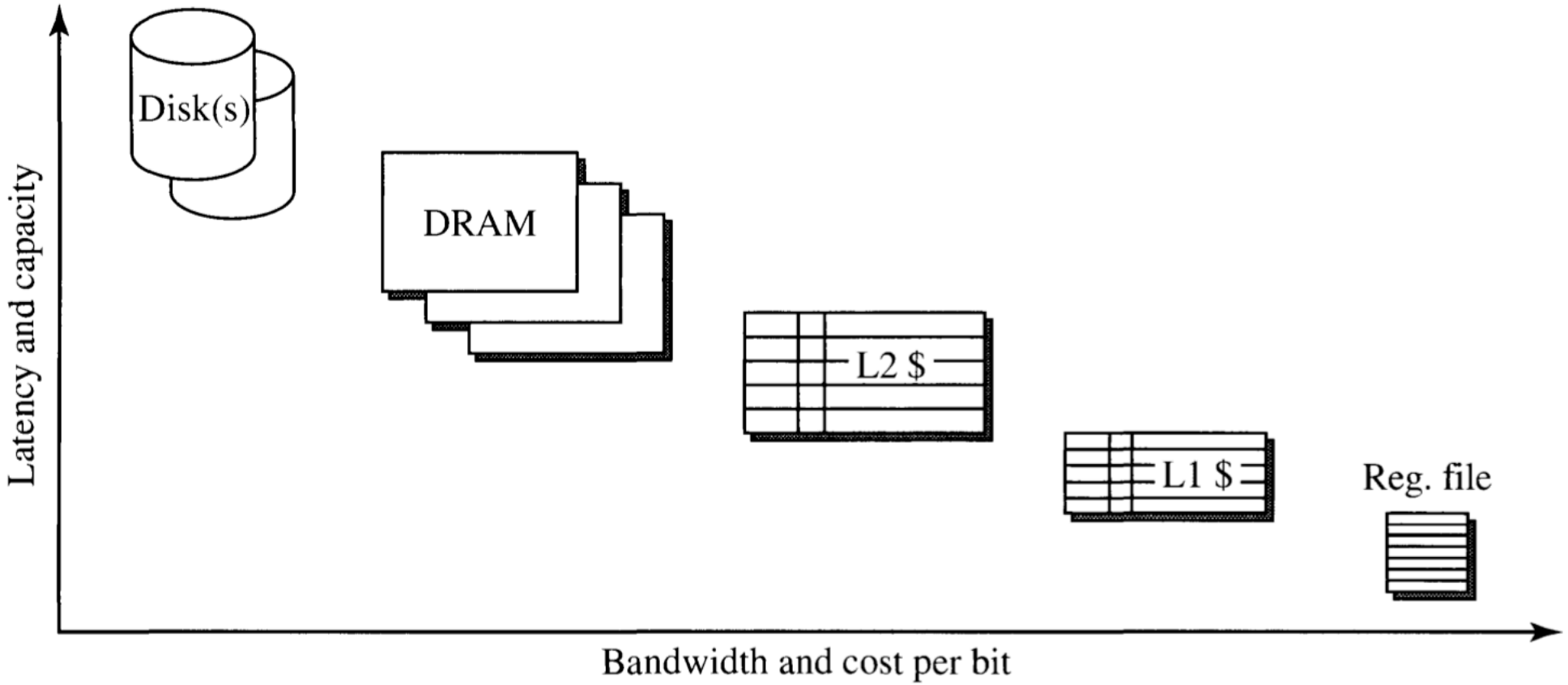
What can we do about it..



What can we do about it..

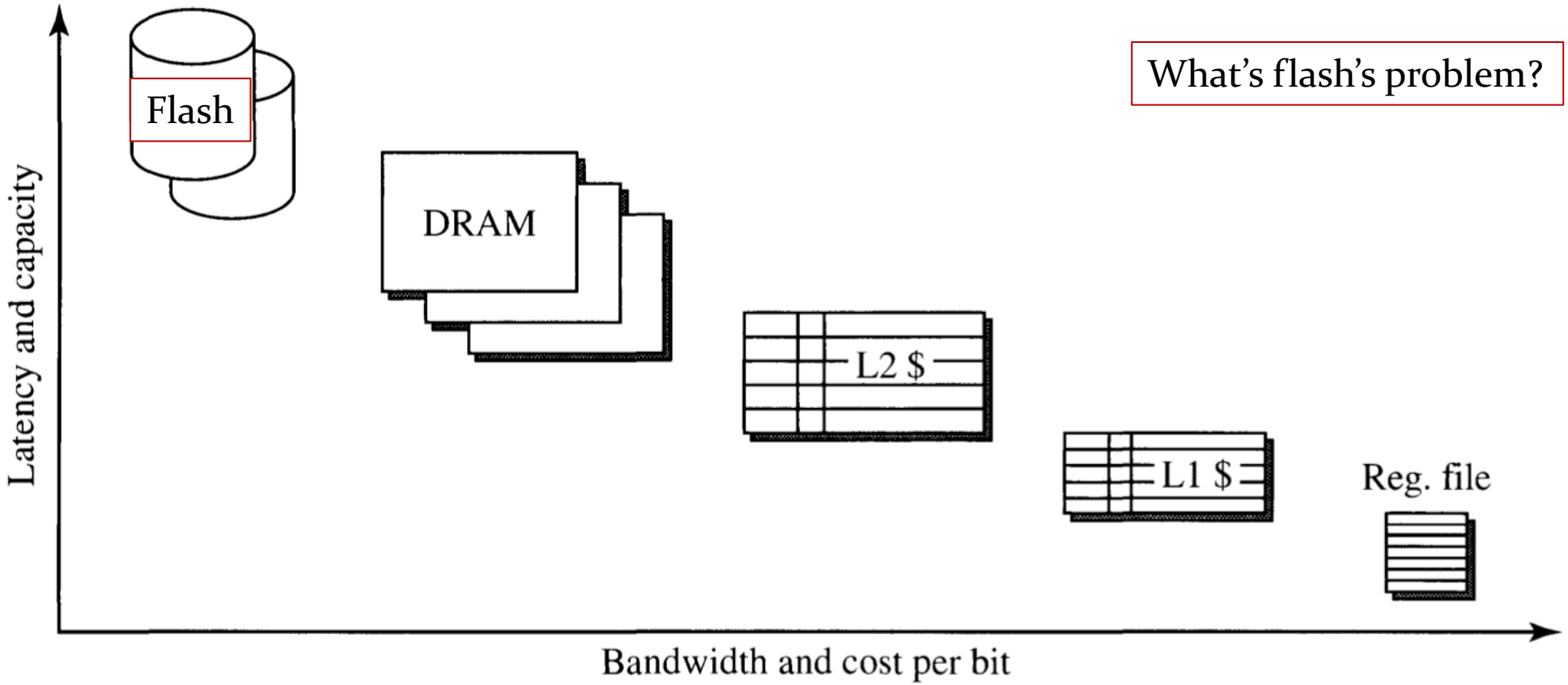


Memory is a technology (physical) problem



We want cheap, high capacity, and fast memory.
We can have 1 of the 3.

Memory is a technology (physical) problem



We want cheap, high capacity, and fast memory.
We can have 1 of the 3.

Solve technology problem with uArch and SW

Programs access a relatively small portion of address space during different phases of execution

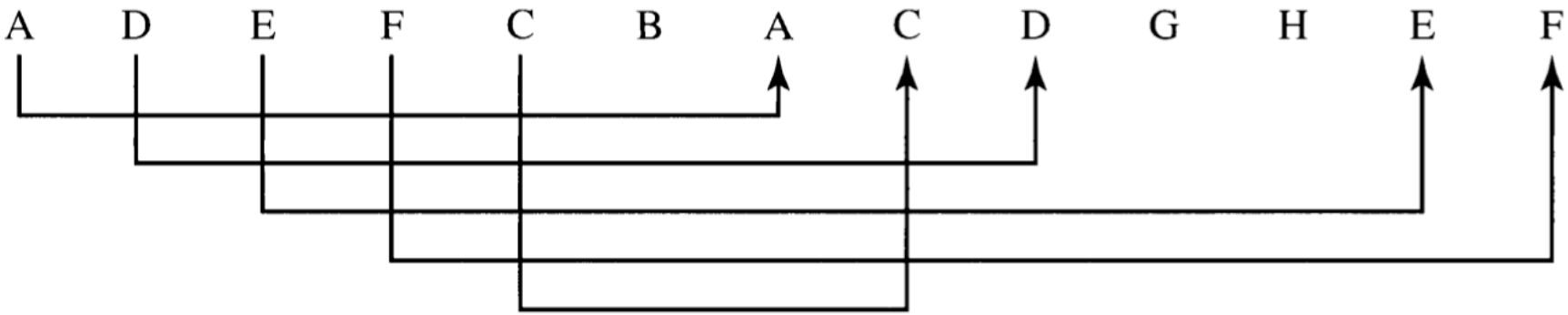
Two Types of Locality:

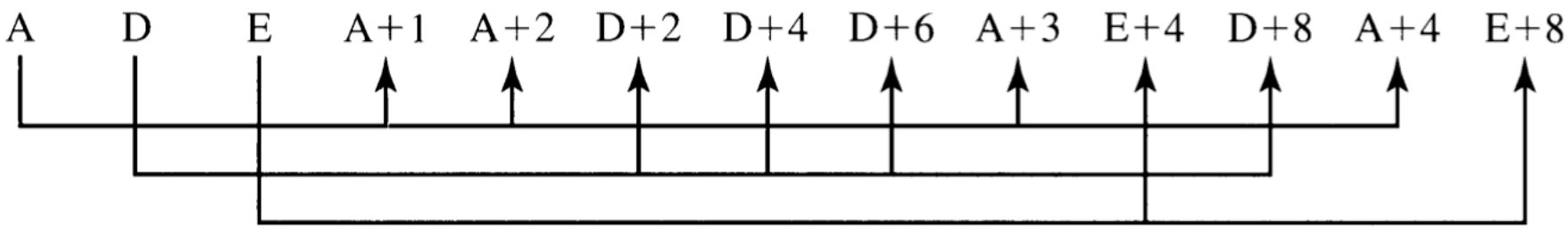
- Temporal Locality (Locality in Time): If an address is referenced, it tends to be referenced again
  - e.g., loops, reuse
- Spatial Locality (Locality in Space): If an address is referenced, neighboring addresses tend to be referenced
  - e.g., array access

Traditionally, HW has relied on locality for speed

Locality is a SW/algo property exploited by machine design.

Access Pattern Exhibiting Locality



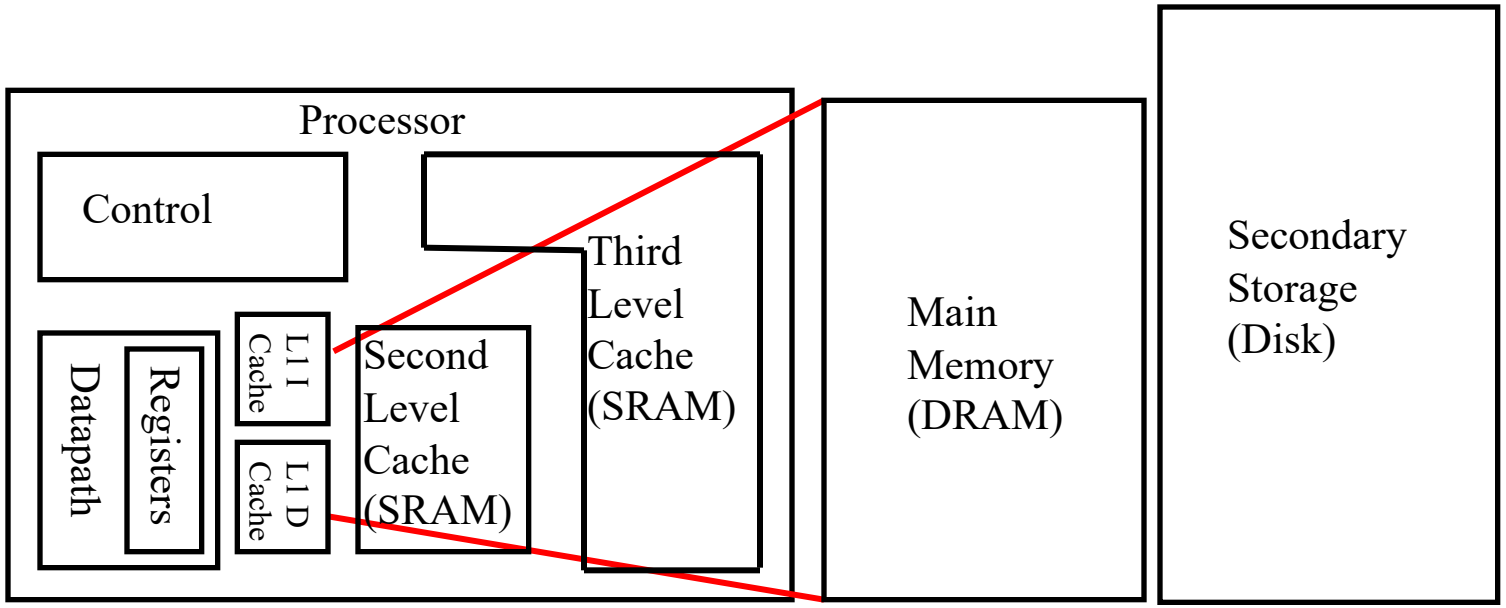


“Memory Hierarchy” or “memory subsystem”

This is how lw and sw are implemented (L1, L2, L3…)

Taking advantage of locality gives the best of both worlds:

- Present the user with as much memory as is available in the cheapest technology
- Provide access at the speed offered by the fastest technology
- Hides complexity!



Cache Terminology

Hit: data is resident in the cache

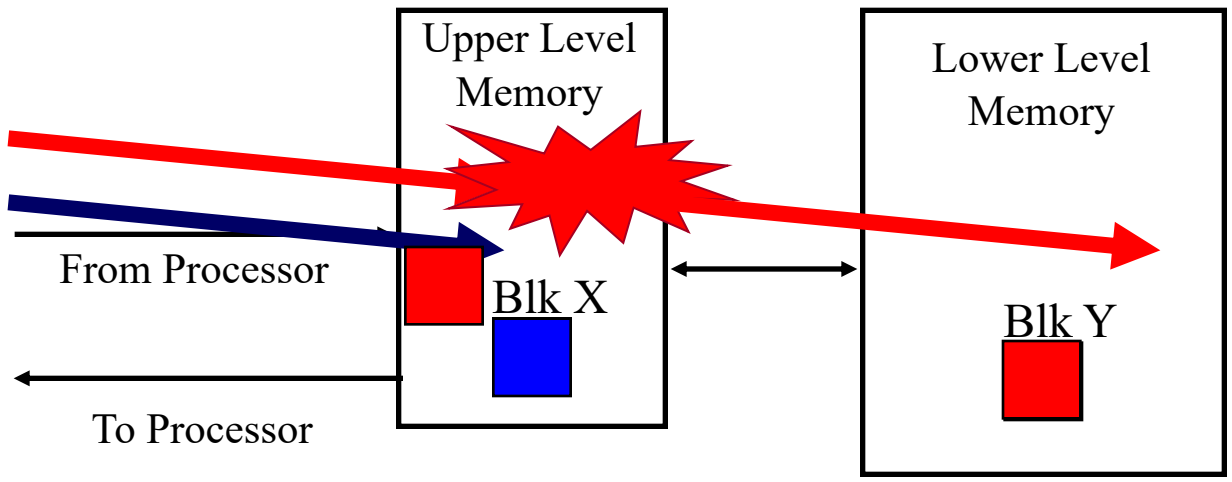
- Hit Rate: the fraction of memory accesses that hit in a level
- Hit Time: Time to access the level (access time + time to determine if hit)

Miss: data needs to be retrieved from a block in the lower level

- Miss Rate = 1 - Hit Rate
- Miss Penalty: Time to replace a block in the upper level +
  Time to deliver the block to the processor

Hit Time << Miss Penalty

Make the common case fast!



Average Memory Access Time (AMAT)

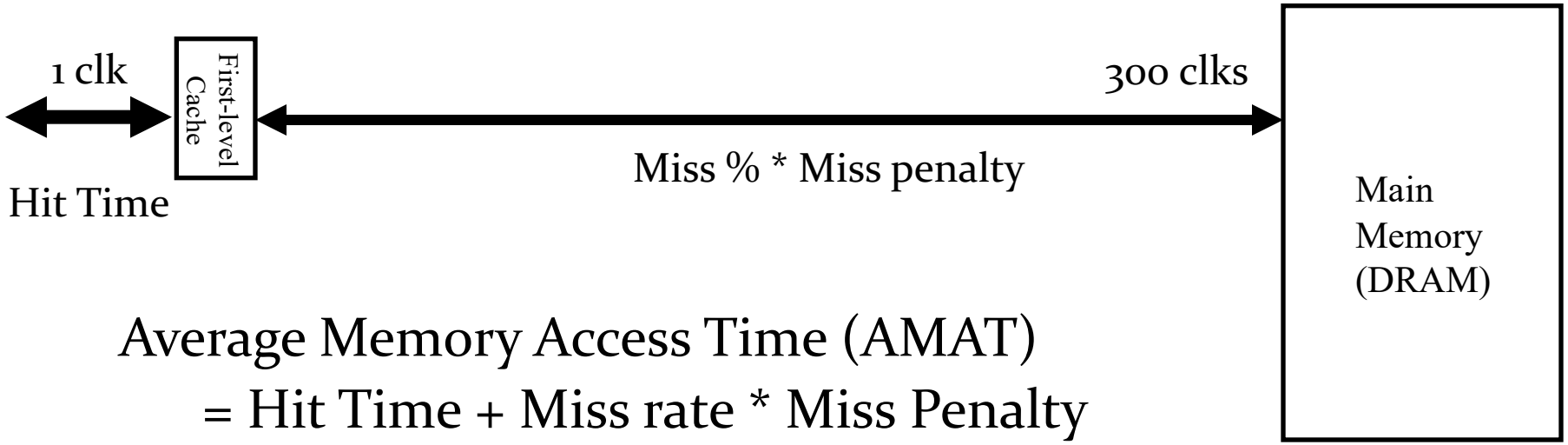
Average memory-access time

- How long does it get to read/write?
- AMAT = Hit time + Miss rate x Miss penalty

Miss penalty: time spent fetching a block from lower memory level

- *access time*: function of latency
- *transfer time*: function of bandwidth b/w levels
  - **Transfer one "cache line/block" at a time**
  - **Transfer at the size of the memory-bus width**

Memory Hierarchy Performance



Average Memory Access Time (AMAT)

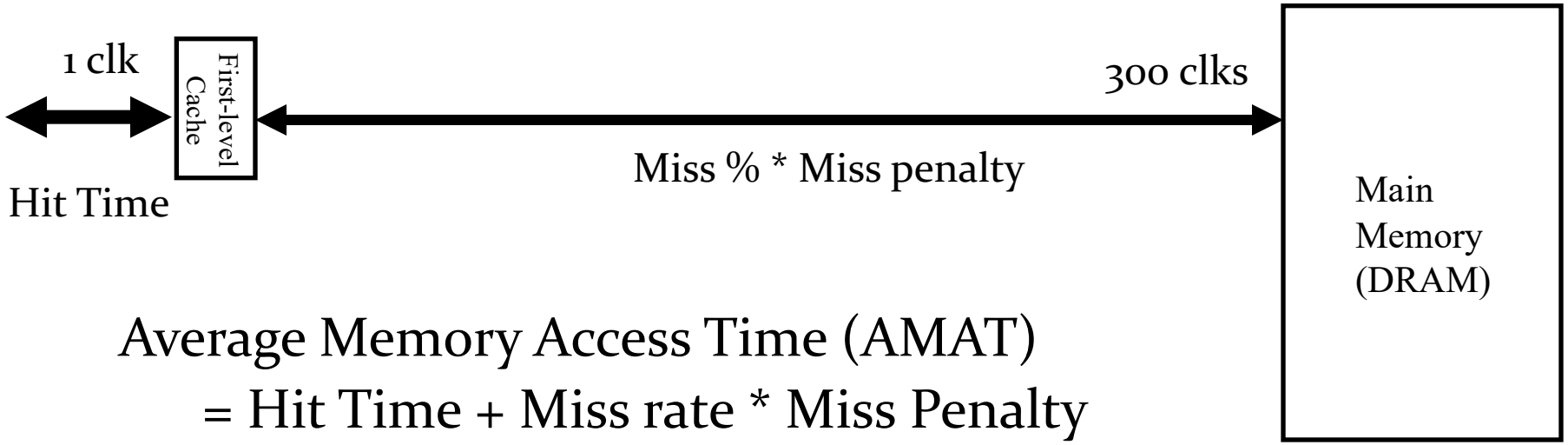
= Hit Time + Miss rate * Miss Penalty

= T_{hit}(L1) + Miss%(L1) * T(memory)

Example:

- Cache Hit = 1 cycle
- Miss rate = 10% = 0.1
- Miss penalty = 300 cycles
- AMAT = ?

Memory Hierarchy Performance



Average Memory Access Time (AMAT)

= Hit Time + Miss rate * Miss Penalty

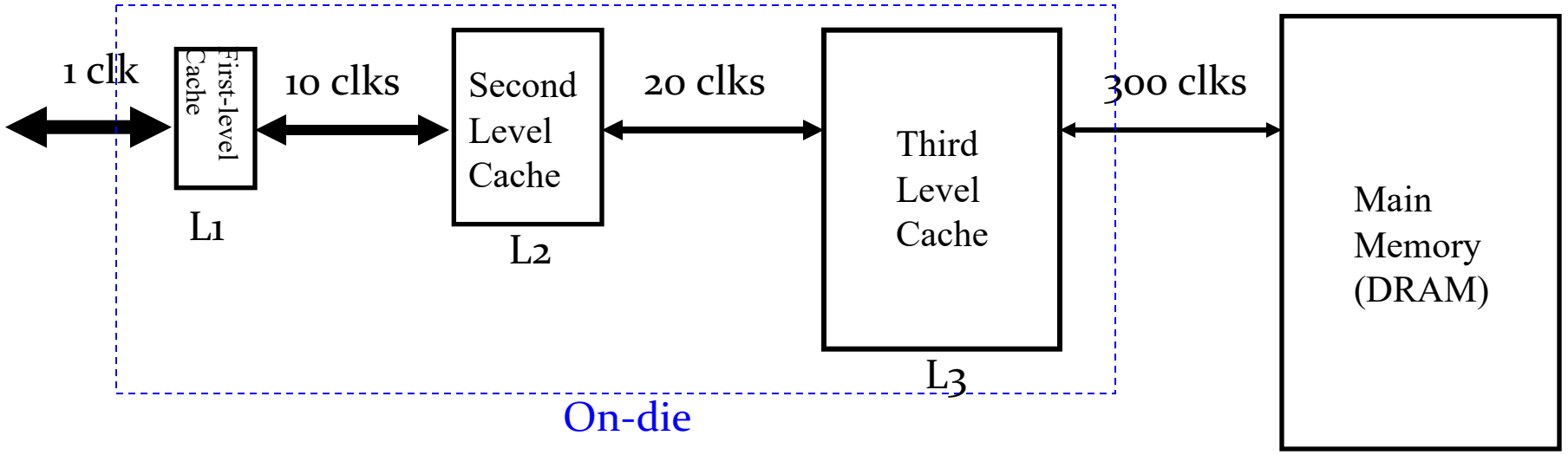
= T_{hit}(L1) + Miss%(L1) * T(memory)

Example:

- Cache Hit = 1 cycle
- Miss rate = 10% = 0.1
- Miss penalty = 300 cycles
- AMAT = 1 + 0.1 * 300 = 31 cycles

Can we improve it?

Reducing Penalty: Multi-Level Cache!



Average Memory Access Time (AMAT)

$= T_{hit}(L1) +$

$MissRate(L1) * [\ T_{hit}(L2) +$

$MissRate(L2) * \{T_{hit}(L3) +$

$MissRate(L3) * T(memory)\ \}\ ]$

AMAT Example

Example:

- Miss rate L1=10%, $T_{hit}(L1)$ = 1 cycle
- Miss rate L2=5%, $T_{hit}(L2)$ = 10 cycles
- Miss rate L3=1%, $T_{hit}(L3)$ = 20 cycles
- T(memory) = 300 cycles

AMAT = ?

- No cache: 300
- L1 only: 31 [9.68 speedup!]
- All levels: 2.115 [14.7x speedup!]

141.8x total speedup.

Caches work very well!

Average Memory Access Time (AMAT)

$= T_{hit}(L1) +$
$MissRate(L1) * [\ T_{hit}(L2) +$
$MissRate(L2) * \{T_{hit}(L3) +$
$MissRate(L3) * T(memory)\ \}\ ]$

AMAT Example

Example:

- Miss rate L1=10%, $T_{hit}(L1)$ = 1 cycle
- Miss rate L2=5%, $T_{hit}(L2)$ = 10 cycles
- Miss rate L3=1%, $T_{hit}(L3)$ = 20 cycles
- T(memory) = 300 cycles

AMAT = 2.115 => 141.8x speedup

Average Memory Access Time (AMAT)

$$= T_{hit}(L1) + MissRate(L1) * [\, T_{hit}(L2) + MissRate(L2) * \{T_{hit}(L3) + MissRate(L3) * T(memory)\ \}\ ]$$

Design question:

a) Cut L1 miss rate in half and double L3 miss rate.

b) Half L2&L3 access time and increase L1 access time by 10%.

Answer: AMAT(a) = 1.565 AMAT(b) = 1.665

Cache Organization

Caches stores 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?

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

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).

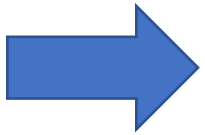
Think about a looking for a book in a library (ordered) verses checking the return pile (who knows!).

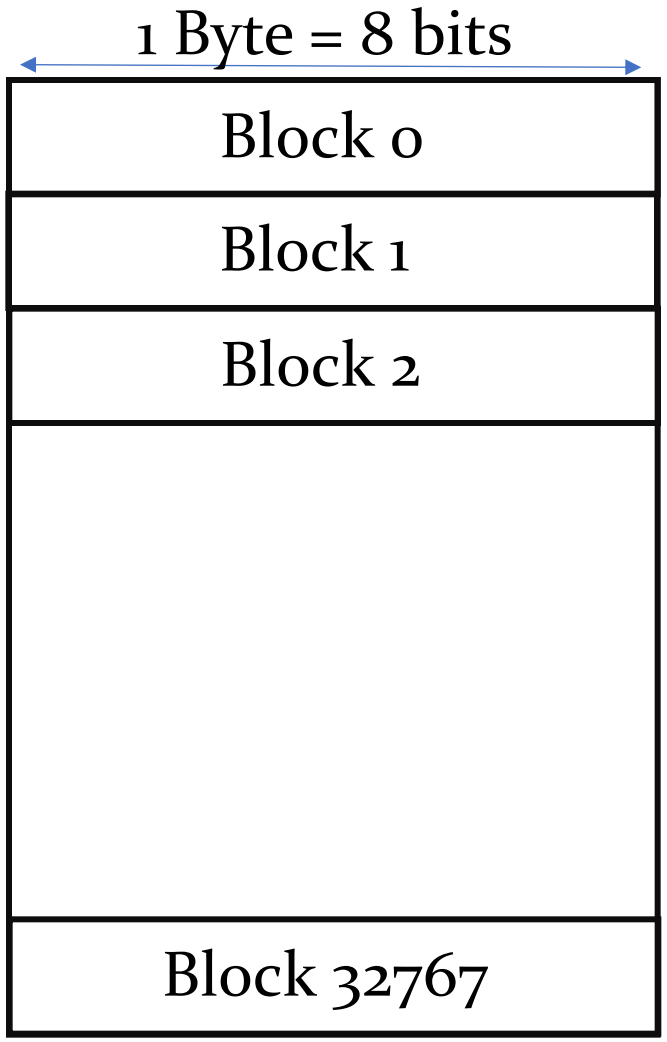
Direct Mapped Cache example

Assume 32 KByte cache with a 1 Byte cache block

- Note that 32KB = 2^{15} Bytes = 32768 Bytes

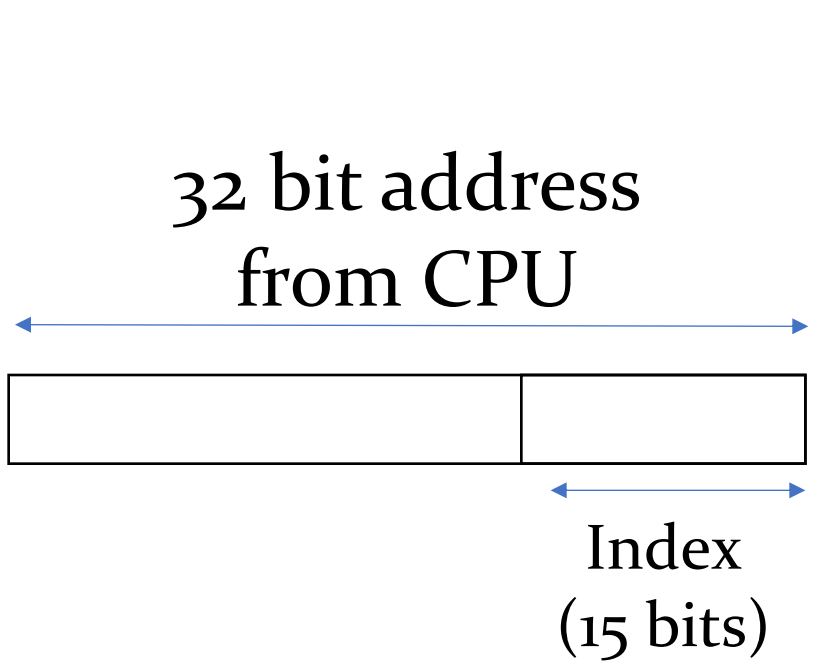
32 bit address from CPU



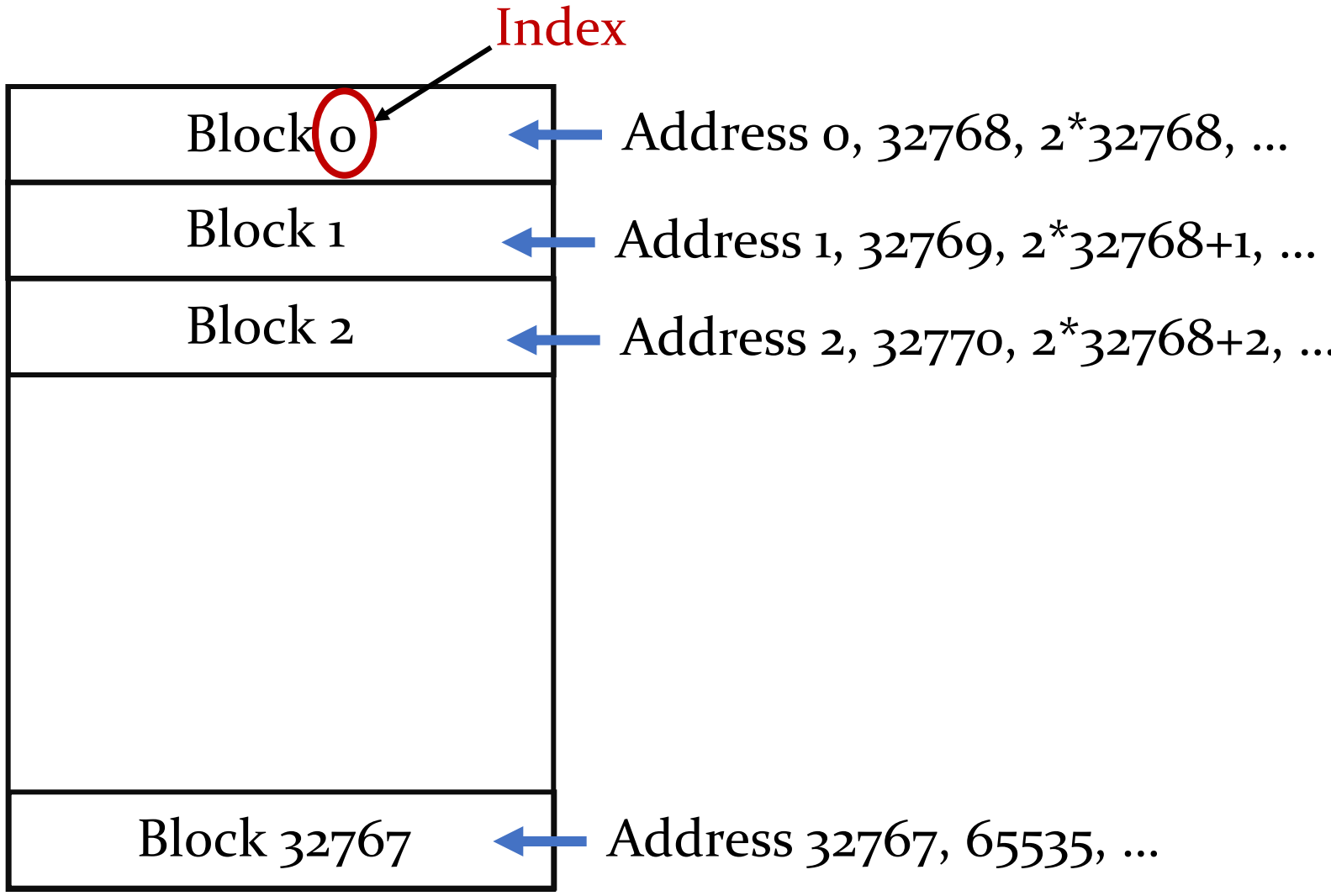


Direct Mapped Cache: Placement

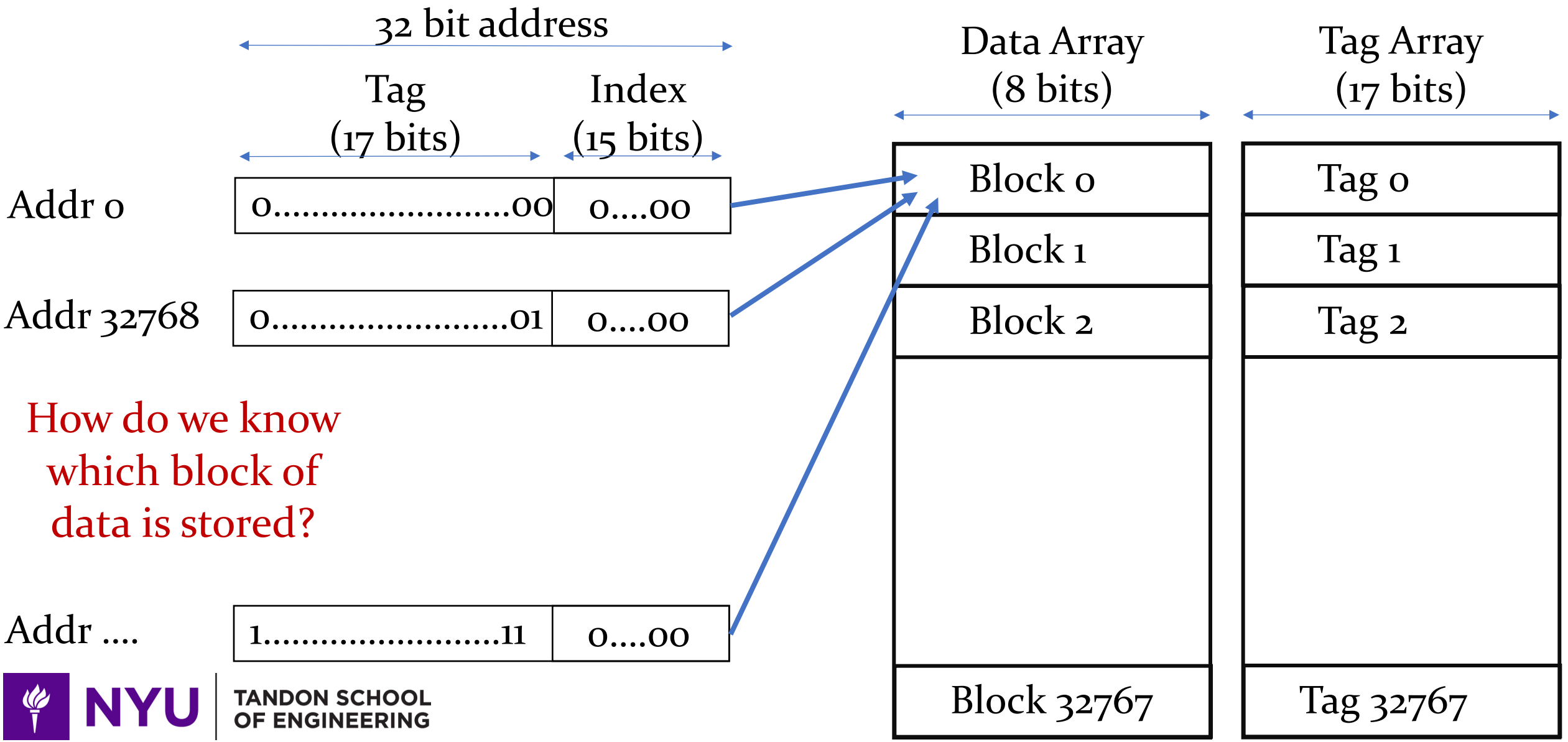
- Each block in the cache has a $\log_2(32768) = 15$ bit "index"



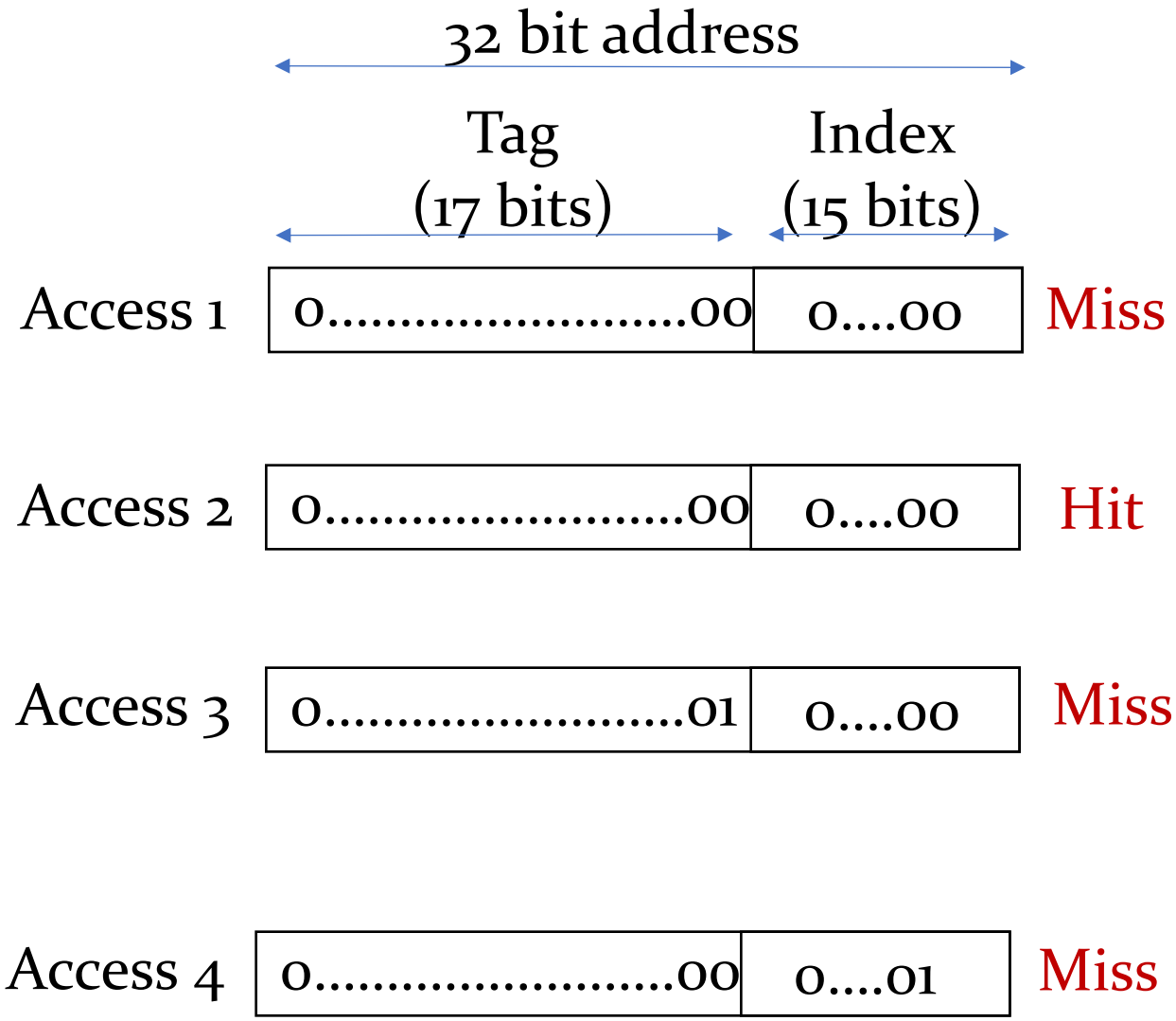
15 LSBs of address (index) used to look up/place block in cache

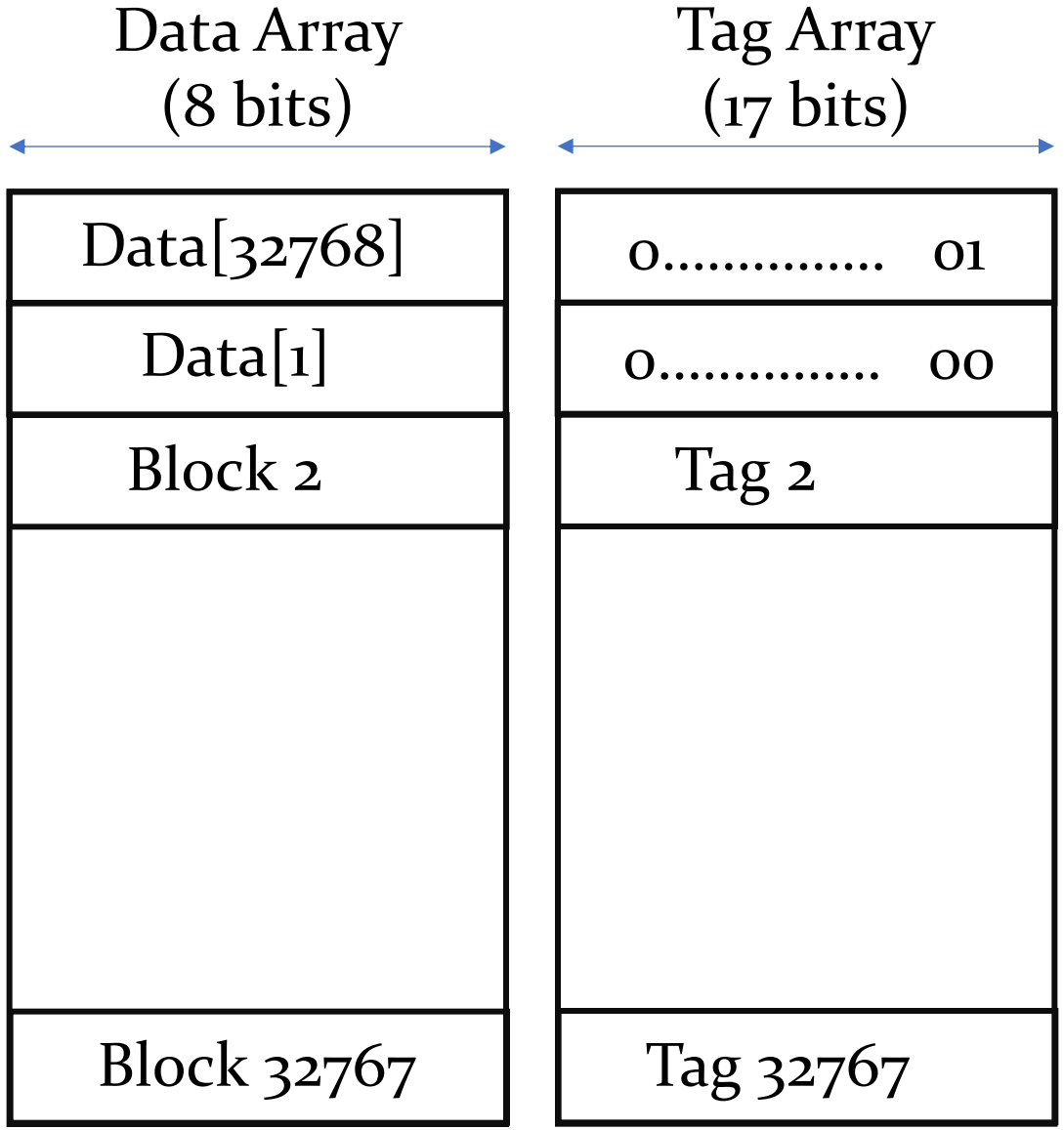


Direct Mapped Cache: Identification

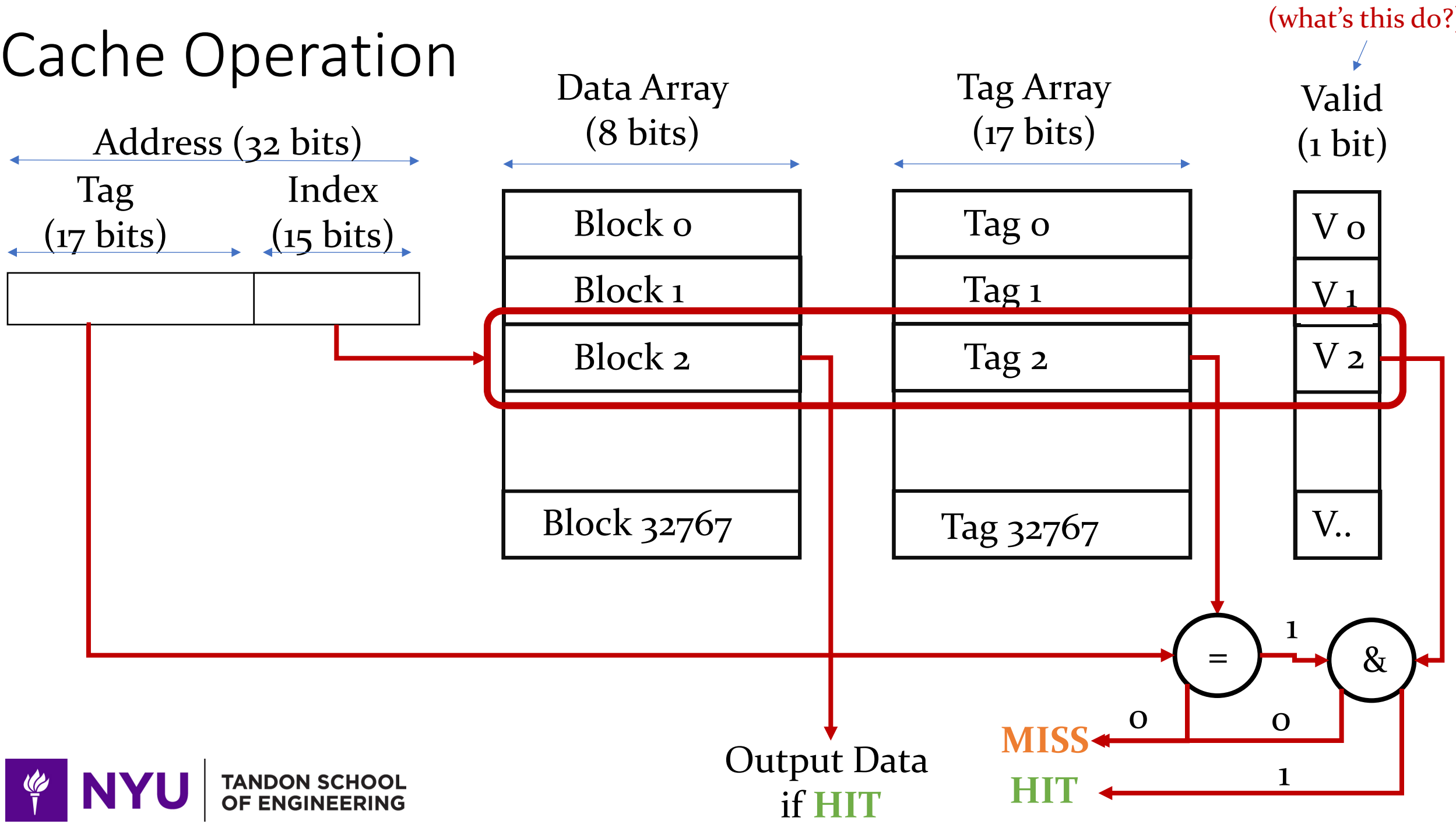


Direct Mapped Cache: Replacement





Cache Operation



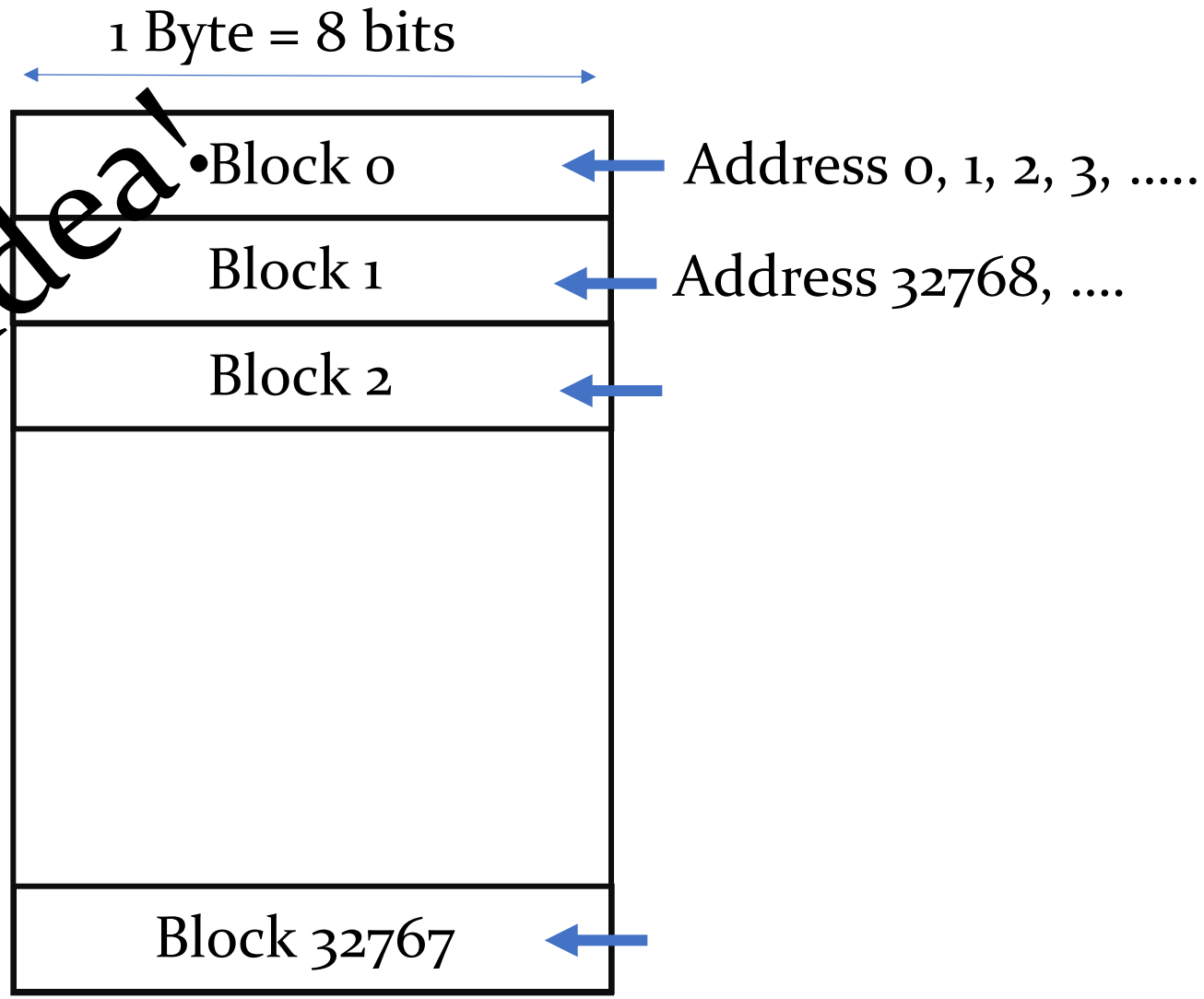
Alternative DM Placement

Assume 32 KByte cache with a 1 Byte cache block

- Use 15 MSBs of address to determine index
- Consider following stream of accesses:

Addr 0, Addr 1, Addr 0, Addr 1,..

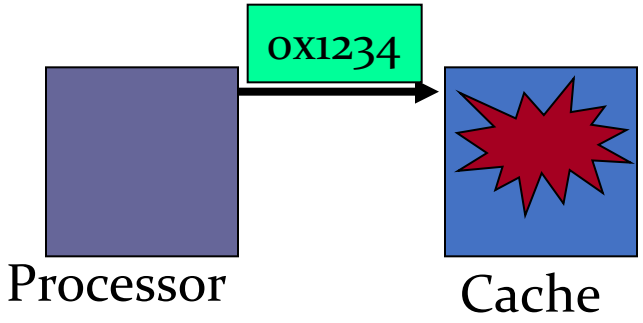
Bad Idea!



Three Cs (Cache Miss Terms)

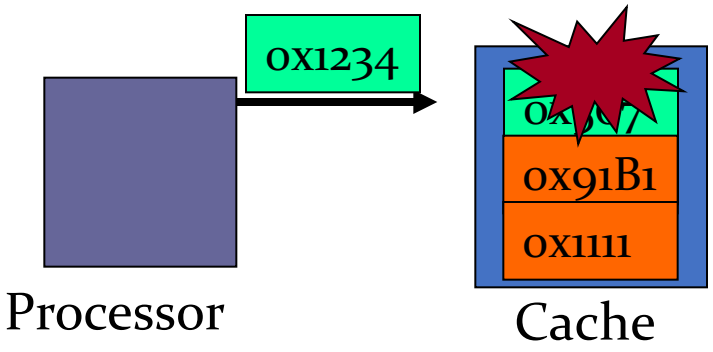
Compulsory Misses:

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



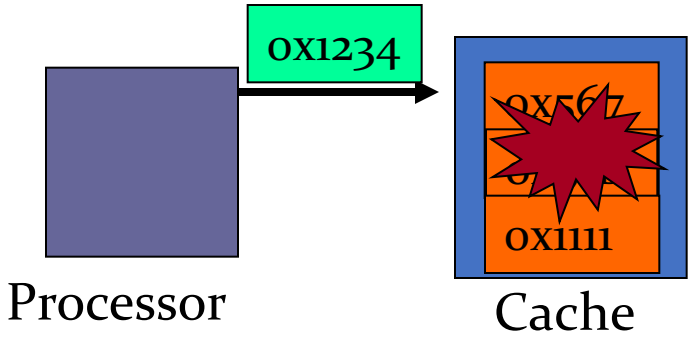
Conflict Misses:

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



Capacity Misses:

- Increase cache size



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

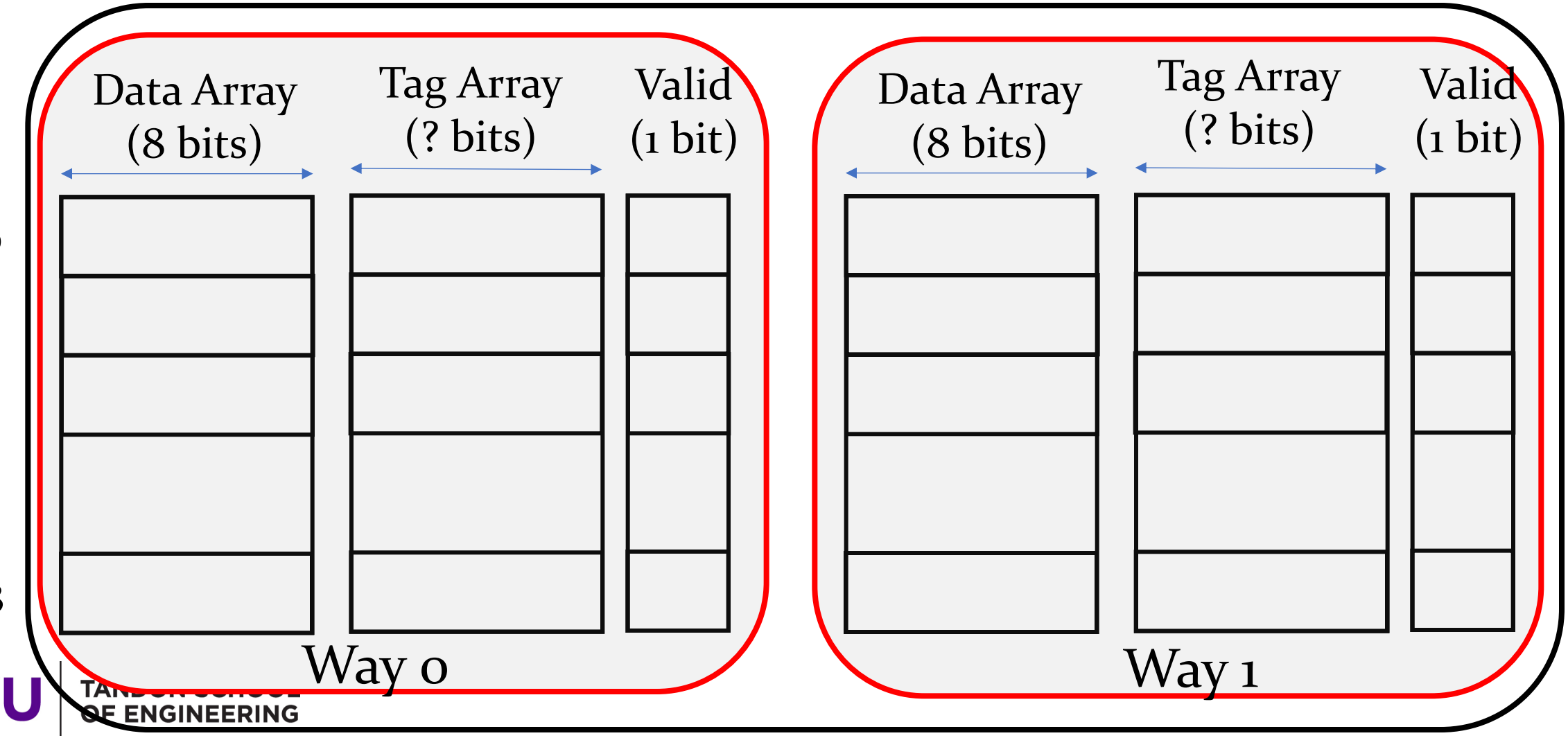
- Sequence: A, B, A, B, A….. results in 100% cache miss rate!

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 …

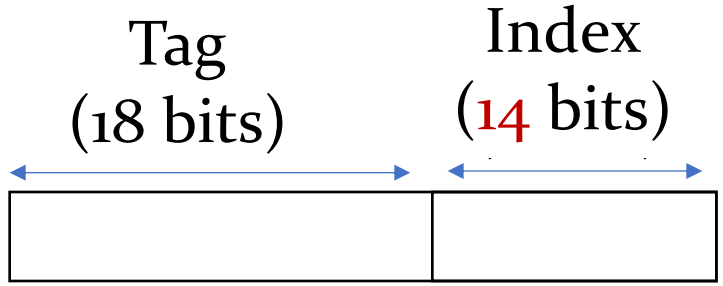
2-Way Set Associative Cache

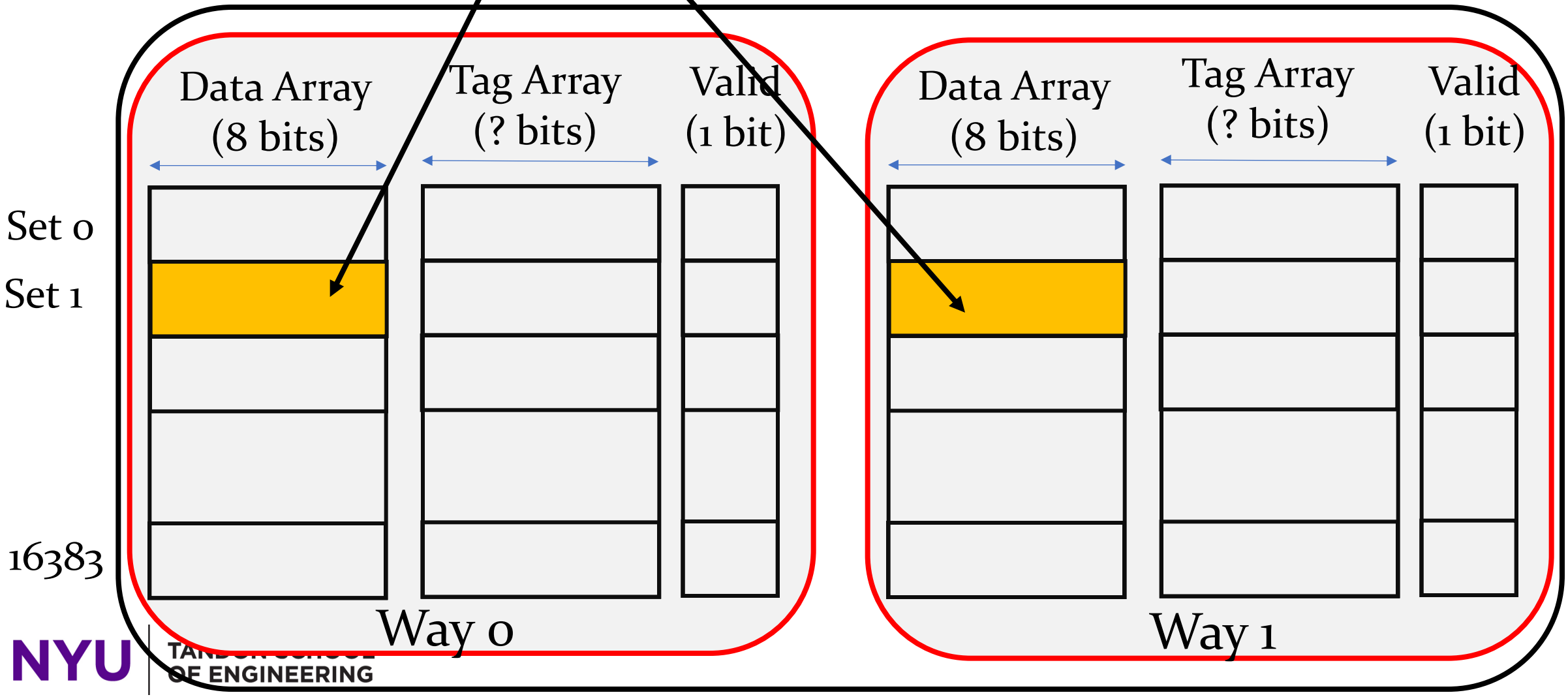
2-way set-associative 32 KB cache with 1 Byte blocks



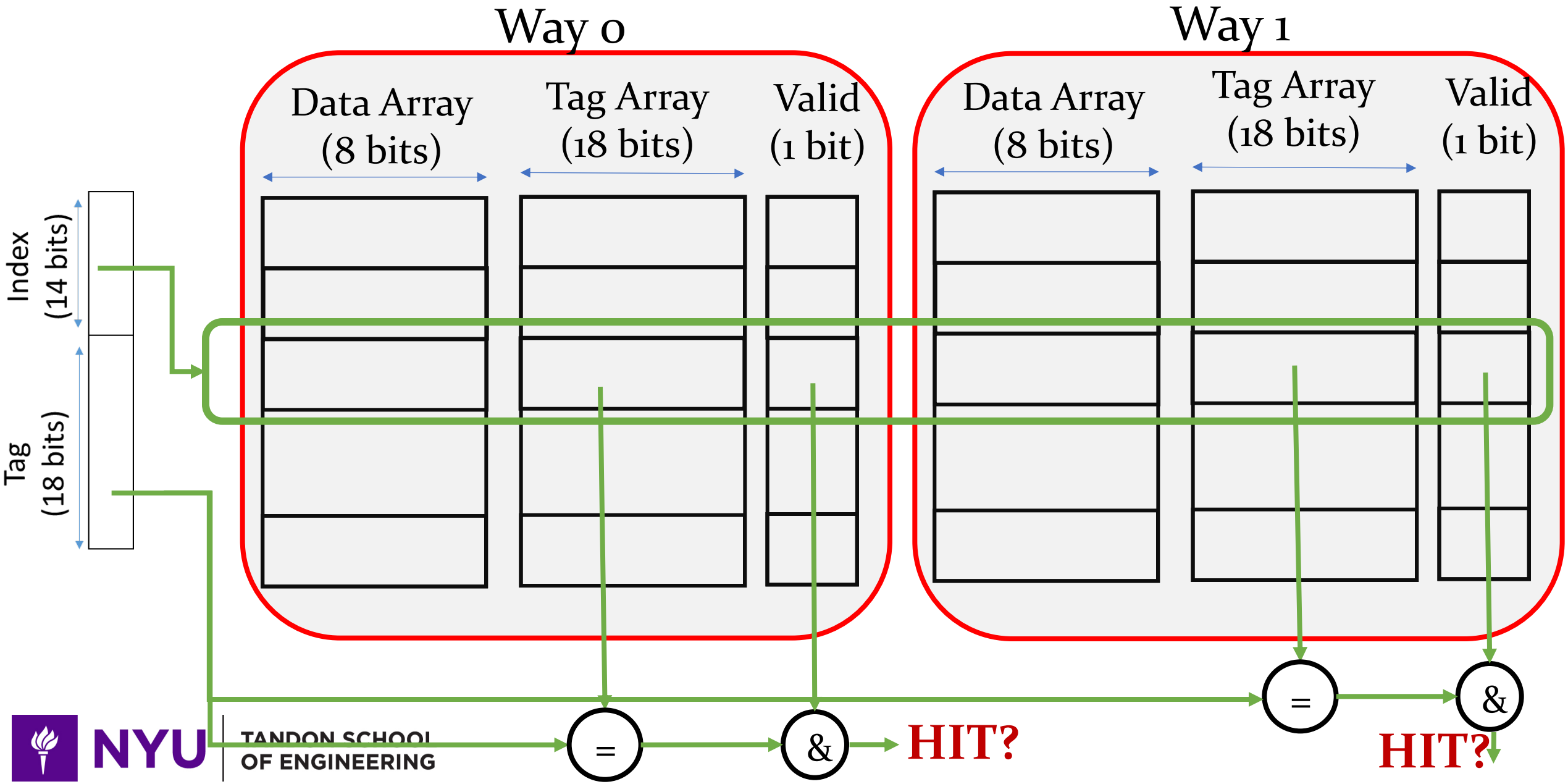
Placement

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



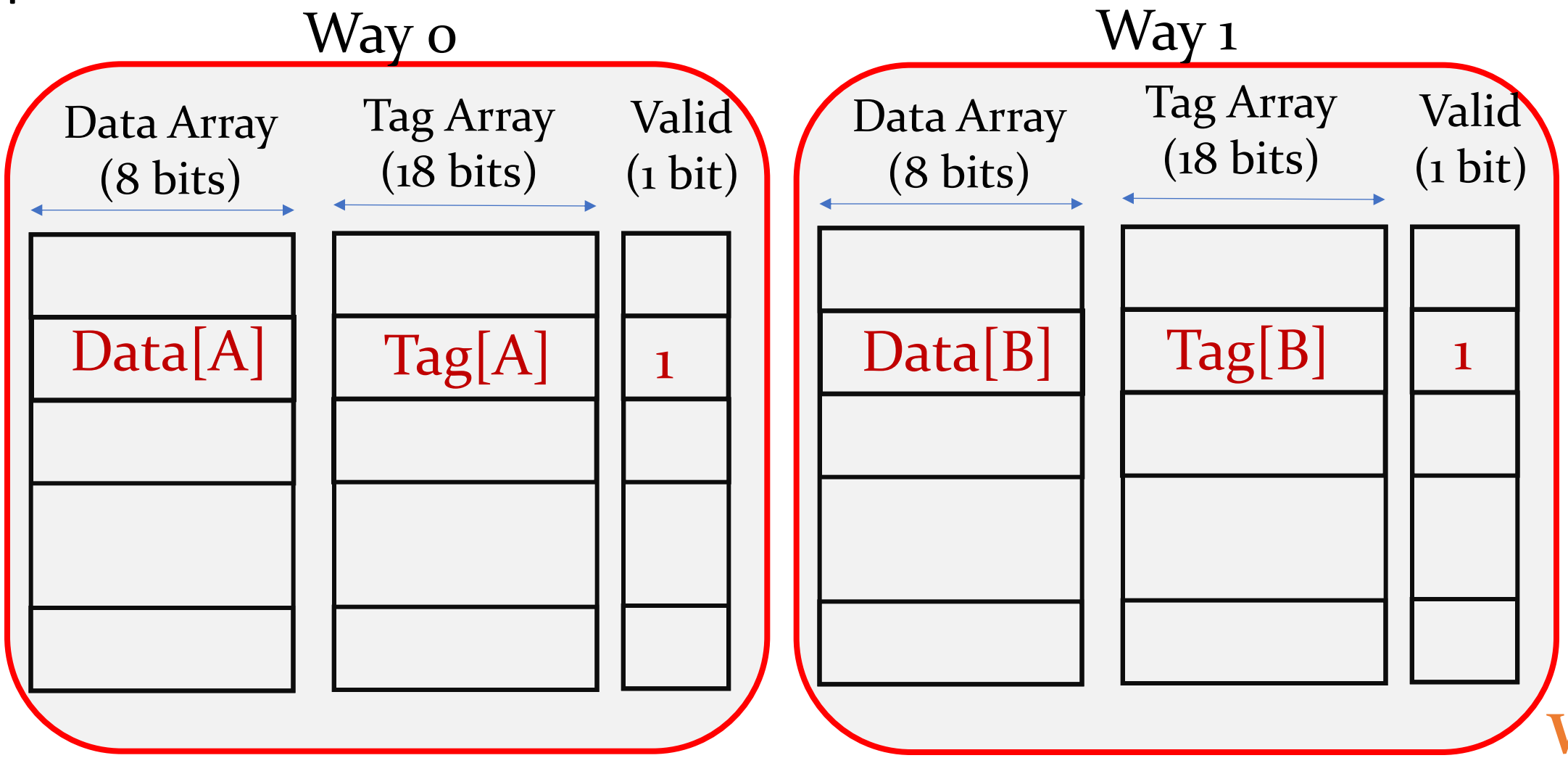


Identification



Replacement

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

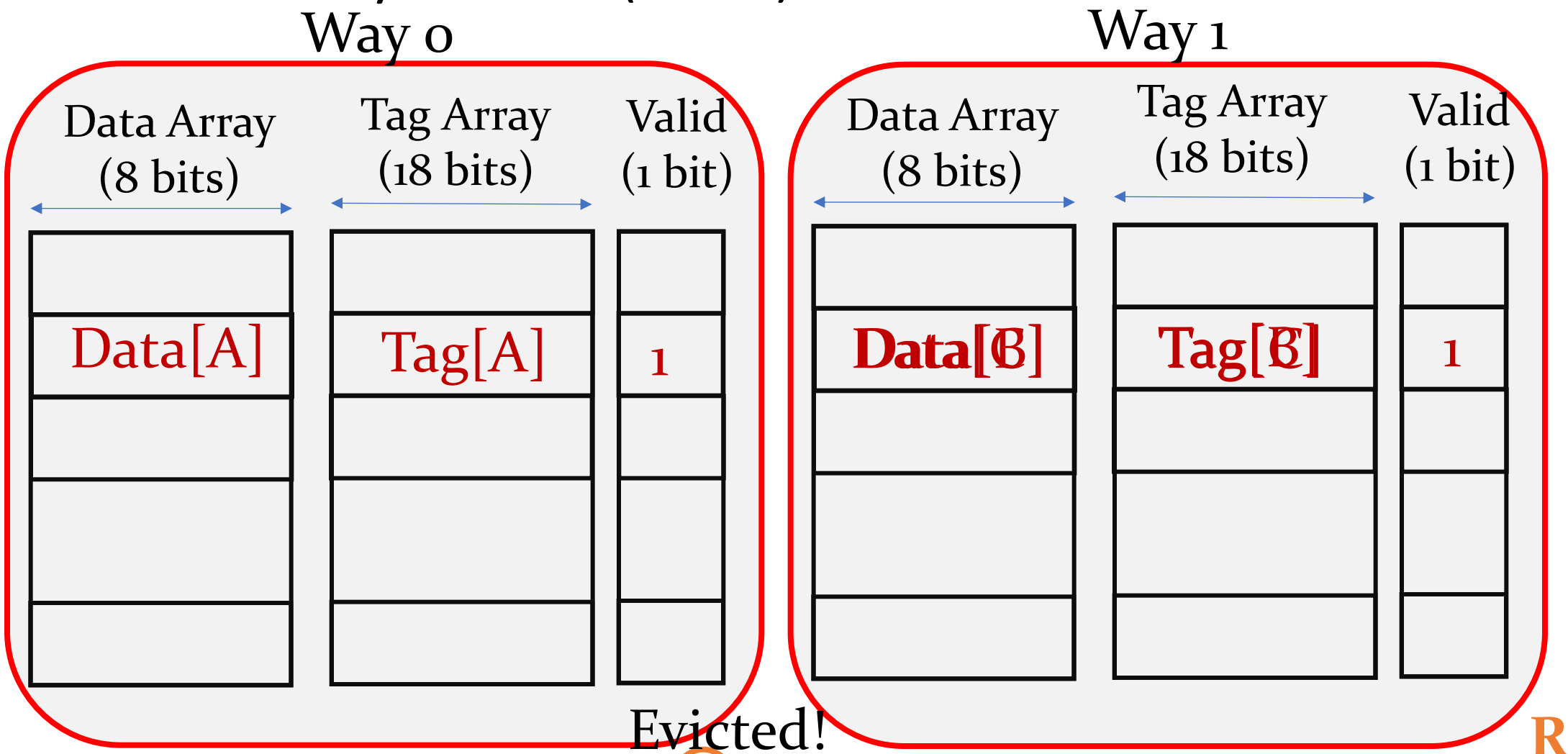


Sequence:	A	B	B	A	A	A	A	C
	M	M	H	H	H	H	H	M

Where should C go?

NYU TANDON SCHOOL OF ENGINEERING

Least Recently Used (LRU)

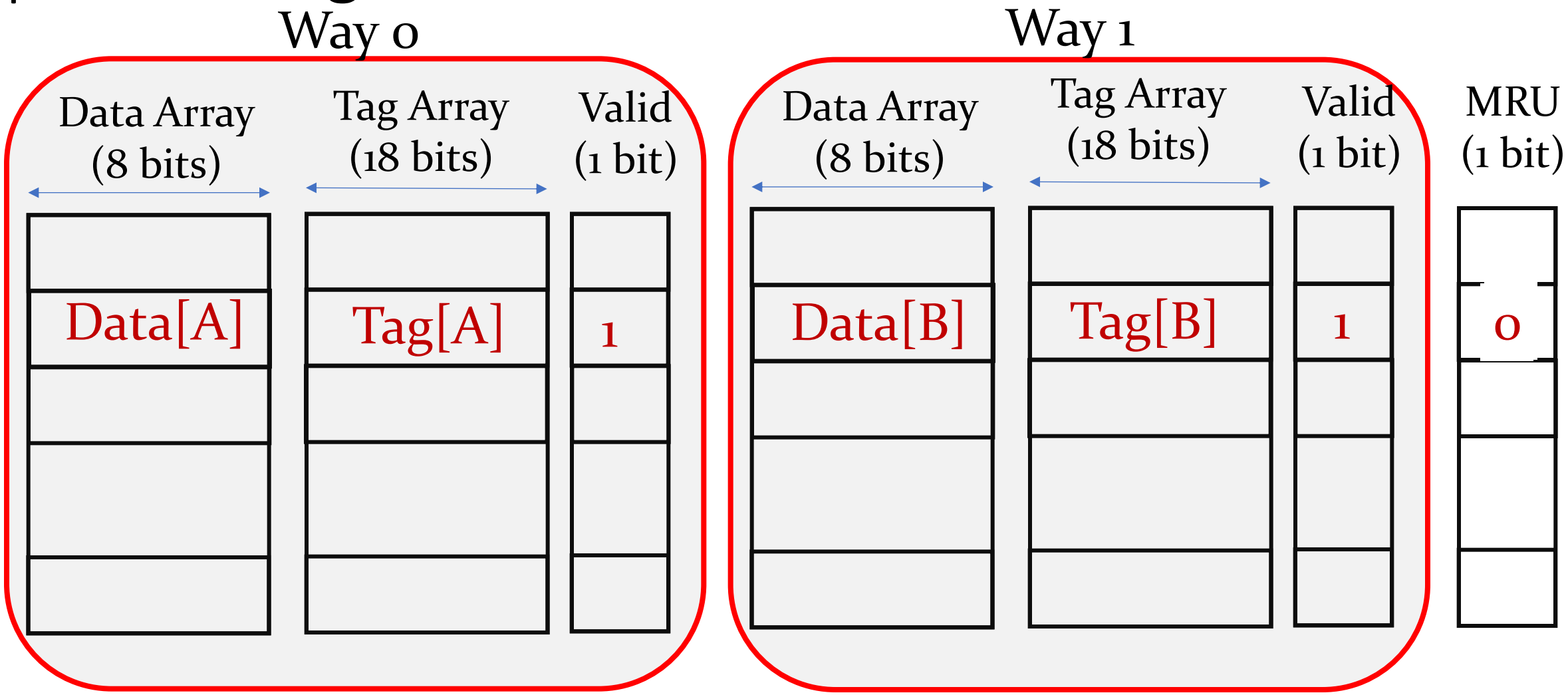


Evicted!

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

Cache Replacement Policy

Random
- Replace a randomly chosen line

FIFO
- Replace the oldest line

LRU (Least Recently Used)
- Replace the least recently used line

NRU (Not Recently Used)
- Replace one of the lines that is not recently used
- Commonly implemented

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?

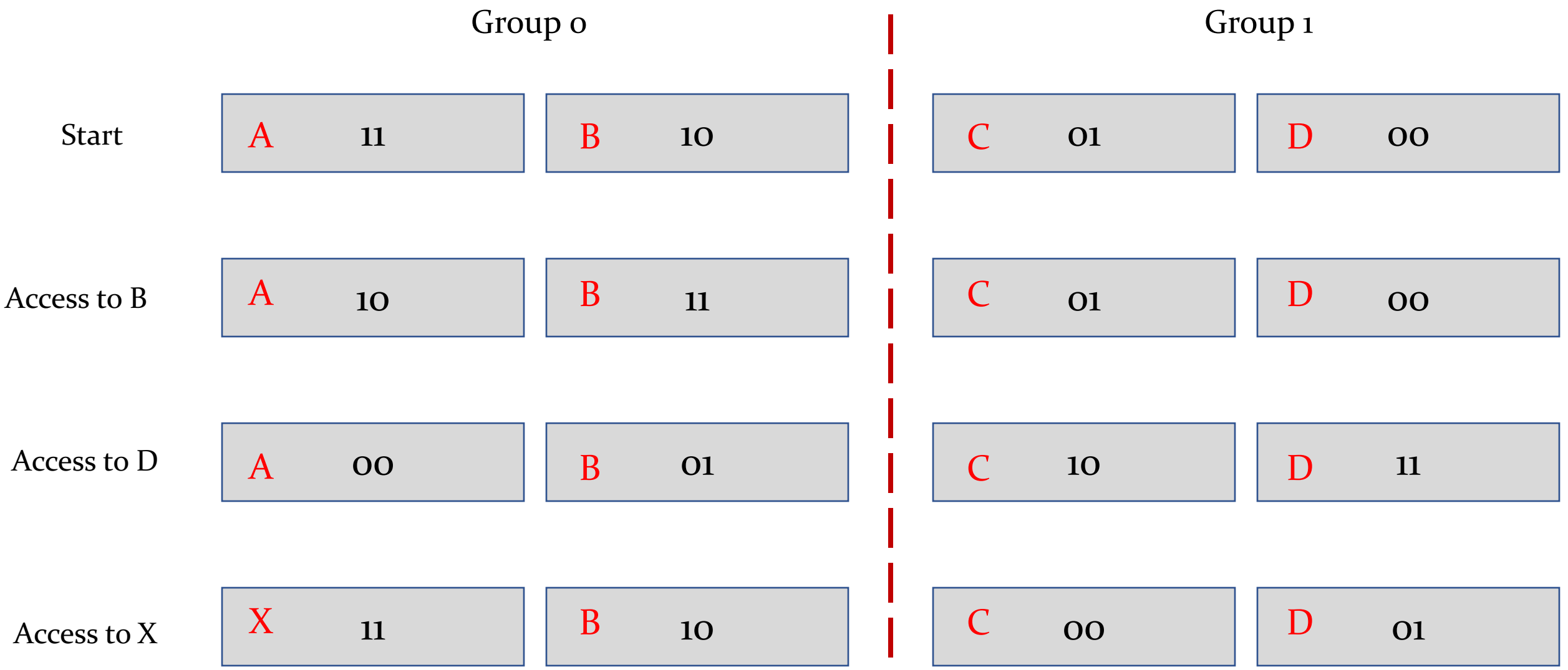
Random isn't all that bad..

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



What locality is this exploiting?

Cache replacement championship!

The Journal of Instruction-Level Parallelism

1st JILP Workshop on Computer Architecture Competitions (JWAC-1):

Cache Replacement Championship

in conjunction with:
ISCA-37 http://isca2010.inria.fr/

The workshop on computer architecture competitions is a forum for holding competitions to evaluate computer architecture research topics. The first workshop is organized around a competition for cache replacement algorithms. The Cache Replacement Championship (CRC) invites contestants to submit their replacement algorithm code to participate in this competition. Contestants will be given a fixed storage budget to implement their best replacement algorithms on a common evaluation framework provided by the organizing committee.

Objective

The goal for this competition is to compare different cache replacement algorithms for a last level cache in a common framework. Replacement algorithms will be evaluated for both private and shared last level caches. The algorithms must be implemented within a fixed storage budget as specified in the competition rules. Submissions will be evaluated based on their performance using the framework provided by the organizing committee. Submissions will be evaluated for two configurations: a single-core configuration with a 1 MB last level cache, and a 4-core configuration with a 4 MB shared last level cache.

https://crc2.ece.tamu.edu/

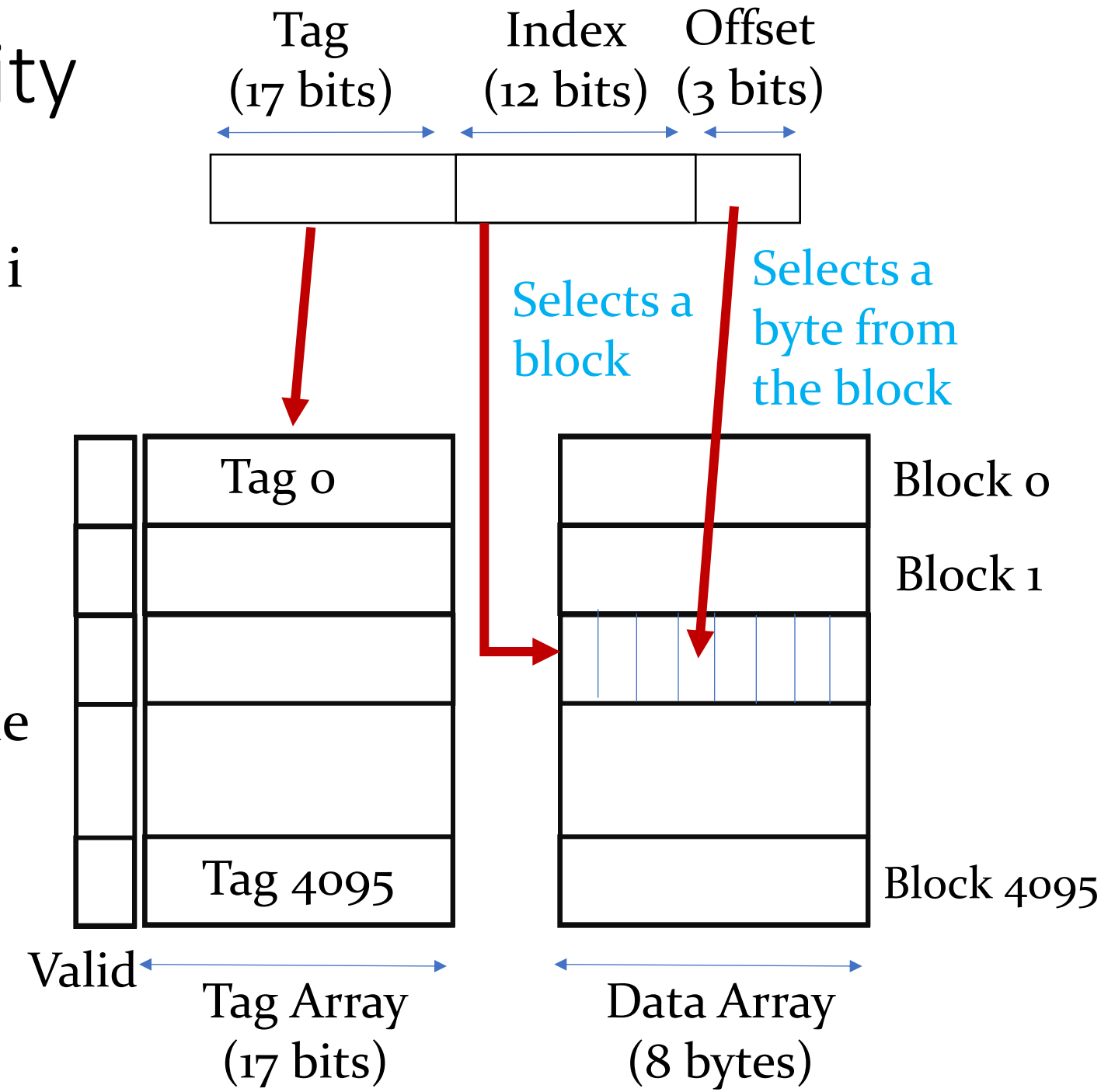
Exploiting Spatial Locality

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!

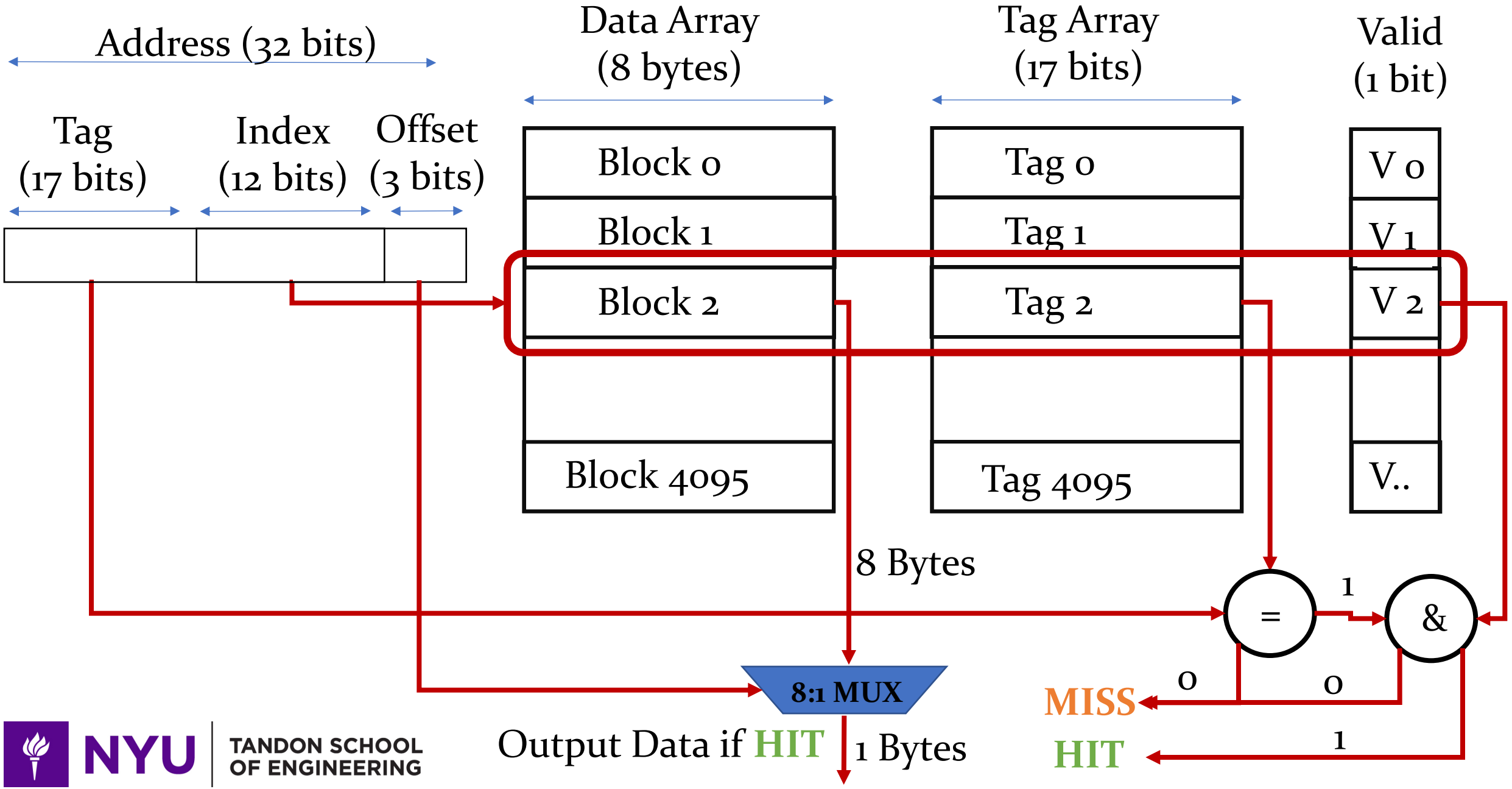
Example: 32KB direct mapped cache with 8 Byte (64 bit) blocks

- i.e., cache has 4096 Byte blocks



Cache Operation

What type of cache is this?



Impact of Block Size

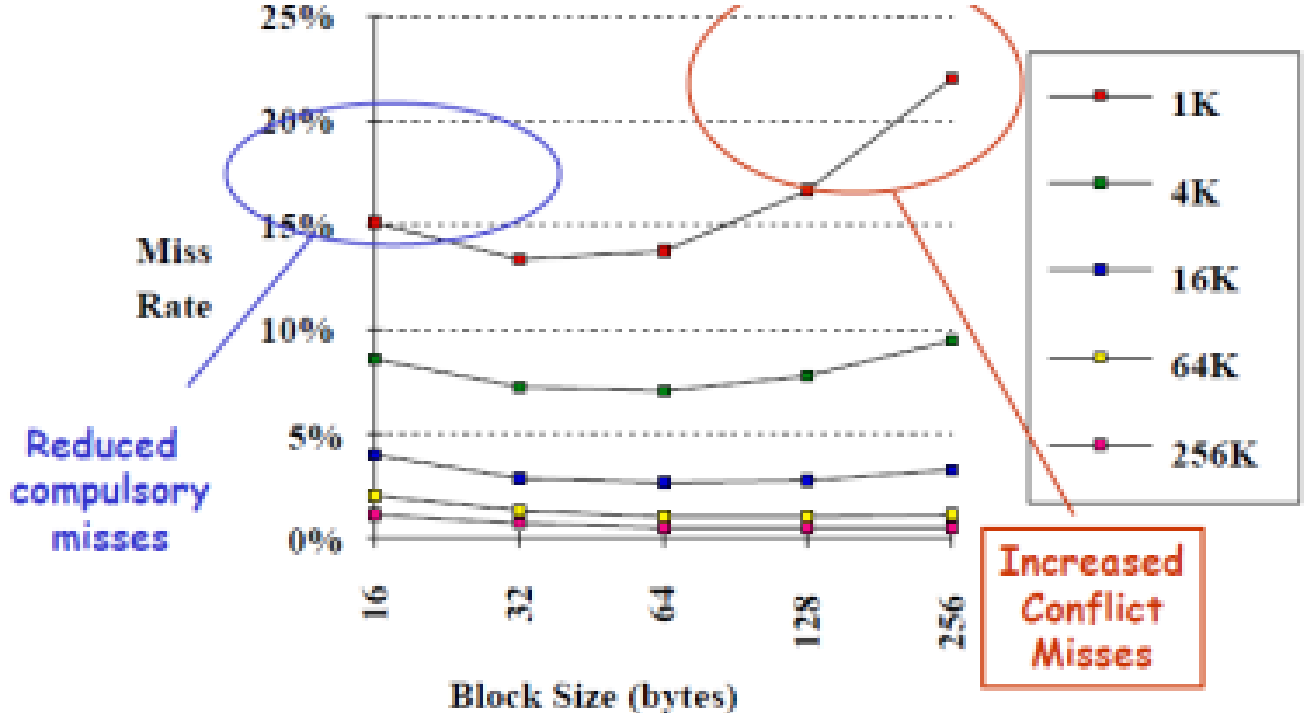
Sequence of addresses: A, A+1, A+2, A+3 …

- 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

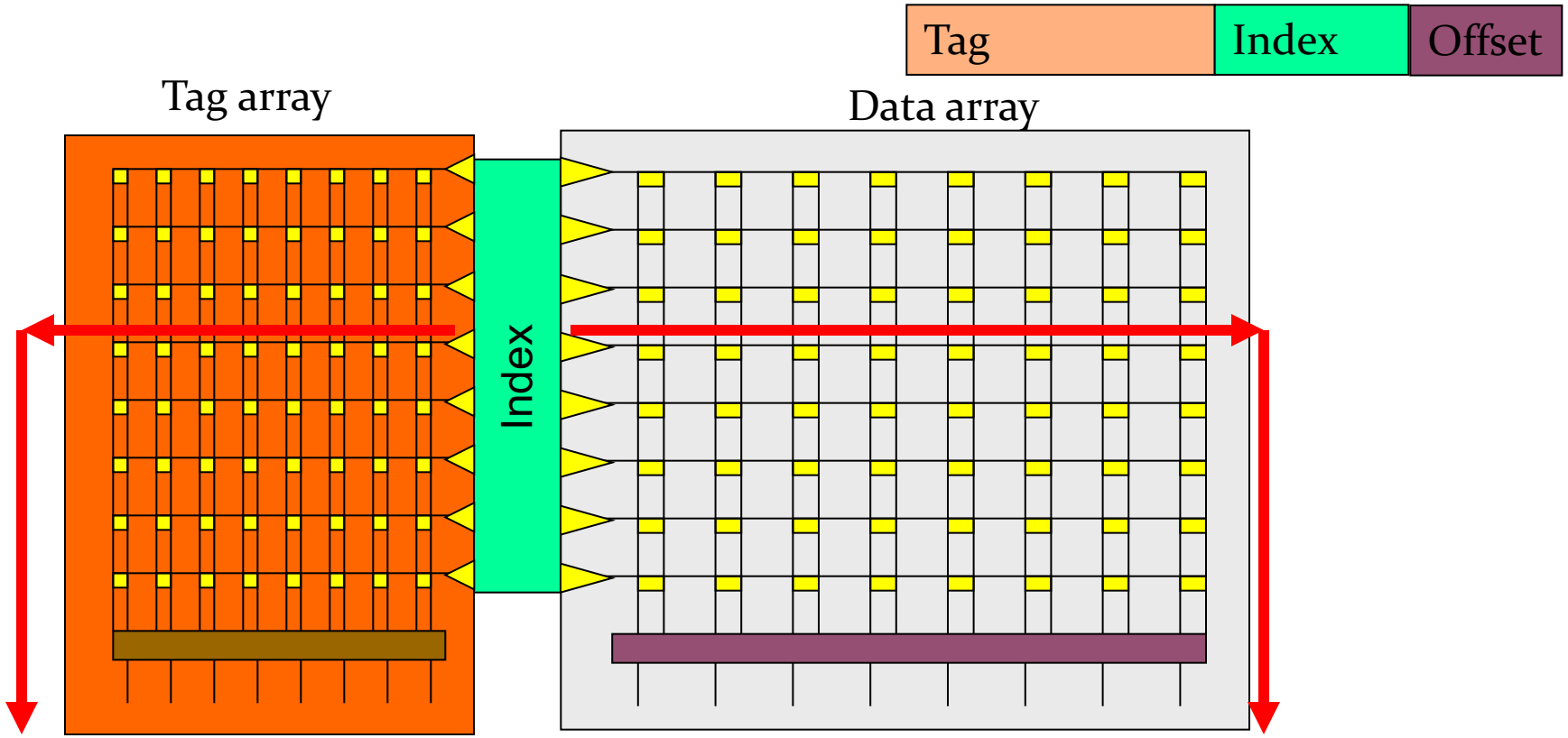
- Fewer number of larger blocks



DM Cache Speed Advantage

Tag and data access happen in parallel

- Faster cache access!

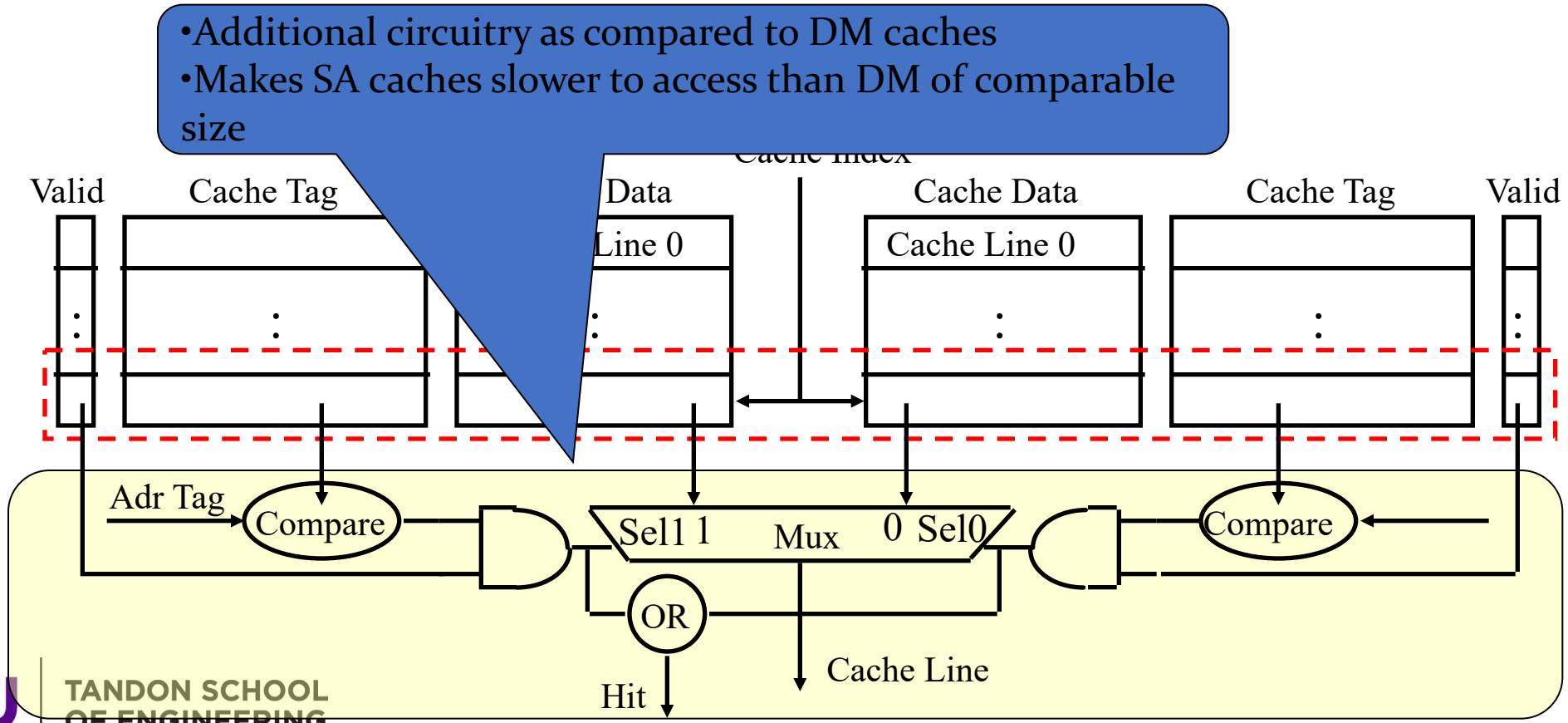


Set Associative Cache (2-way)

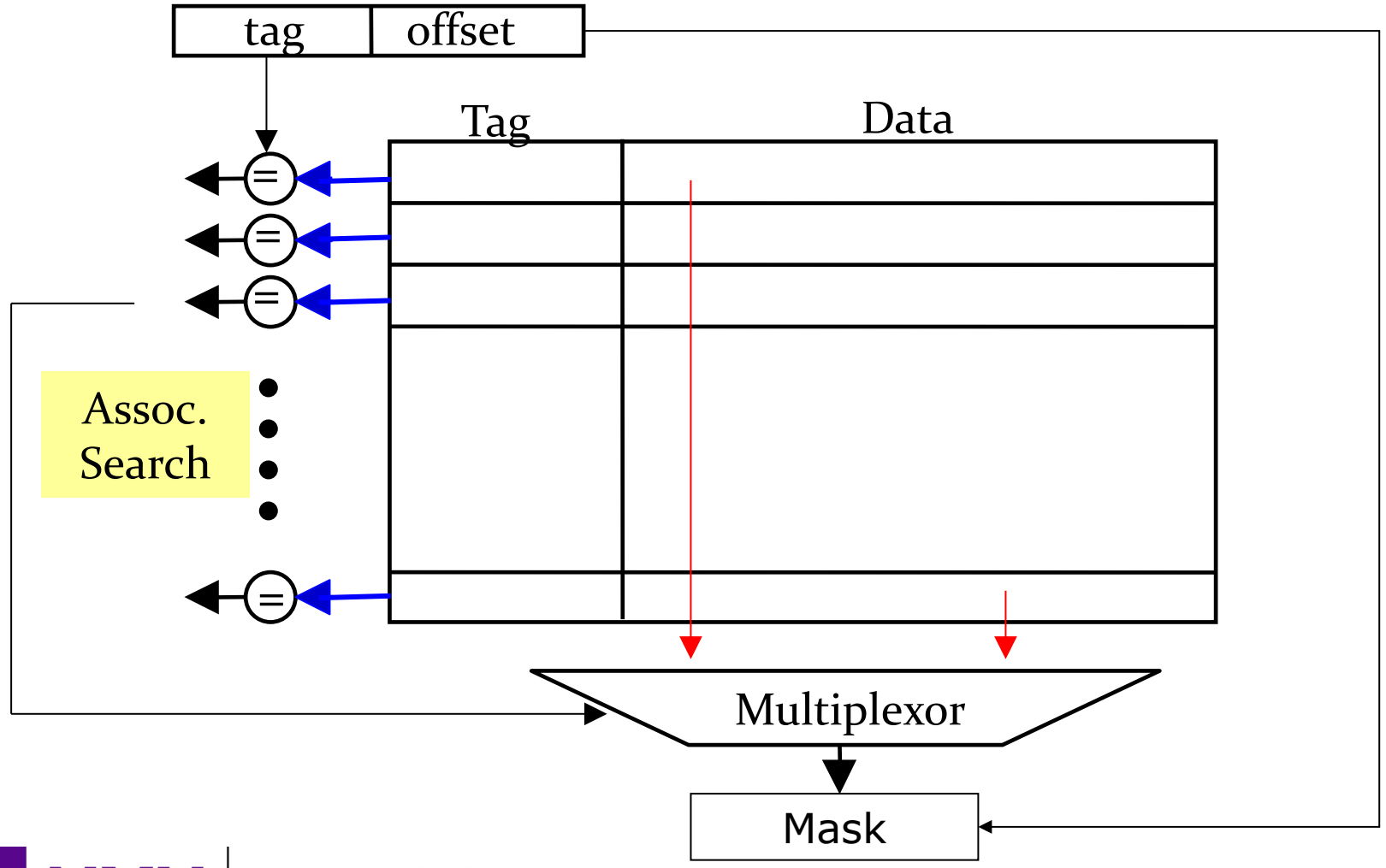
Cache index selects a “set” from the cache

The two tags in the set are compared in parallel

Data is selected based on the tag result



Fully Associative Cache



What are the tradeoffs here?

Pro:
Data can go anywhere!
Implies few conflict misses

Con:
1) Data can go anywhere!
have to check everywhere
2) How to handle replacement?

Example of Caches

Assume:

- addresses are 32b
- a 32KB cache
- line size is 64bytes

Direct Mapped (DM) cache Tag size?

Set Assoc. (SA) 16-way, Tag size?

Fully Assoc. (FA), Tag size?

Example of Caches

Assume:

- addresses are 32b
- a 32KB cache
- line size is 64bytes

Offset = 6b

Lines = 32kb / 64B = 2^15 / 2^6 = 2^9

Sets = # Lines / # Ways = 2^9 / 2^4 = 2^5

Direct Mapped (DM) cache Tag size? 32 – 6 (offset) – 9 (index) = 17 tag bits

Set Assoc. (SA) 16-way, Tag size? 32 – 6 (offset) – 5 (index) = 21 tag bits

Fully Assoc. (FA), Tag size? 32 – 6 (offset) – 0 (index) = 26 tag bits

// How do DM/FA work in Sets equation?

Write Policies

What should we do on a cache store/write access

- 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

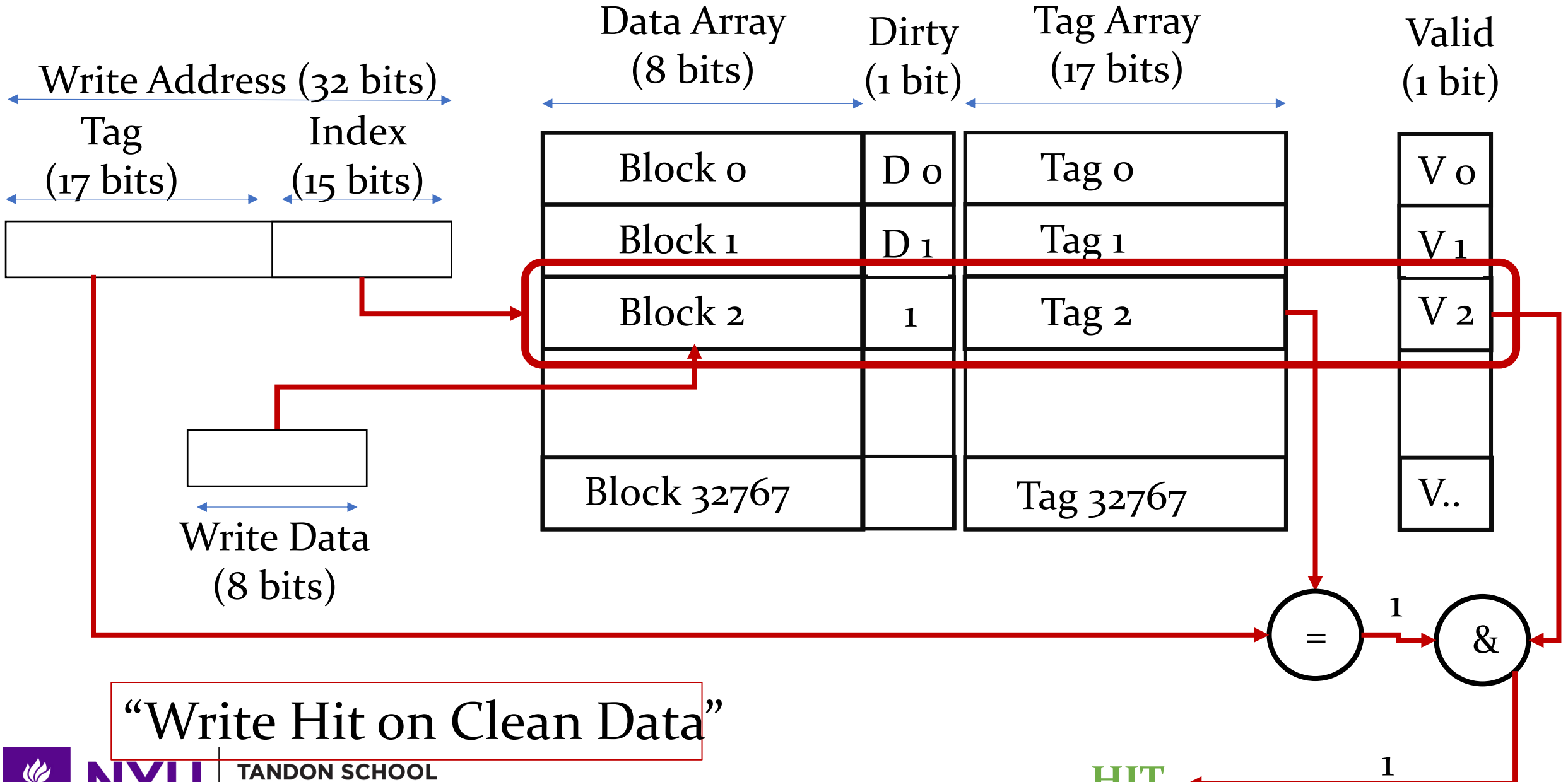
Pros

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

Cons

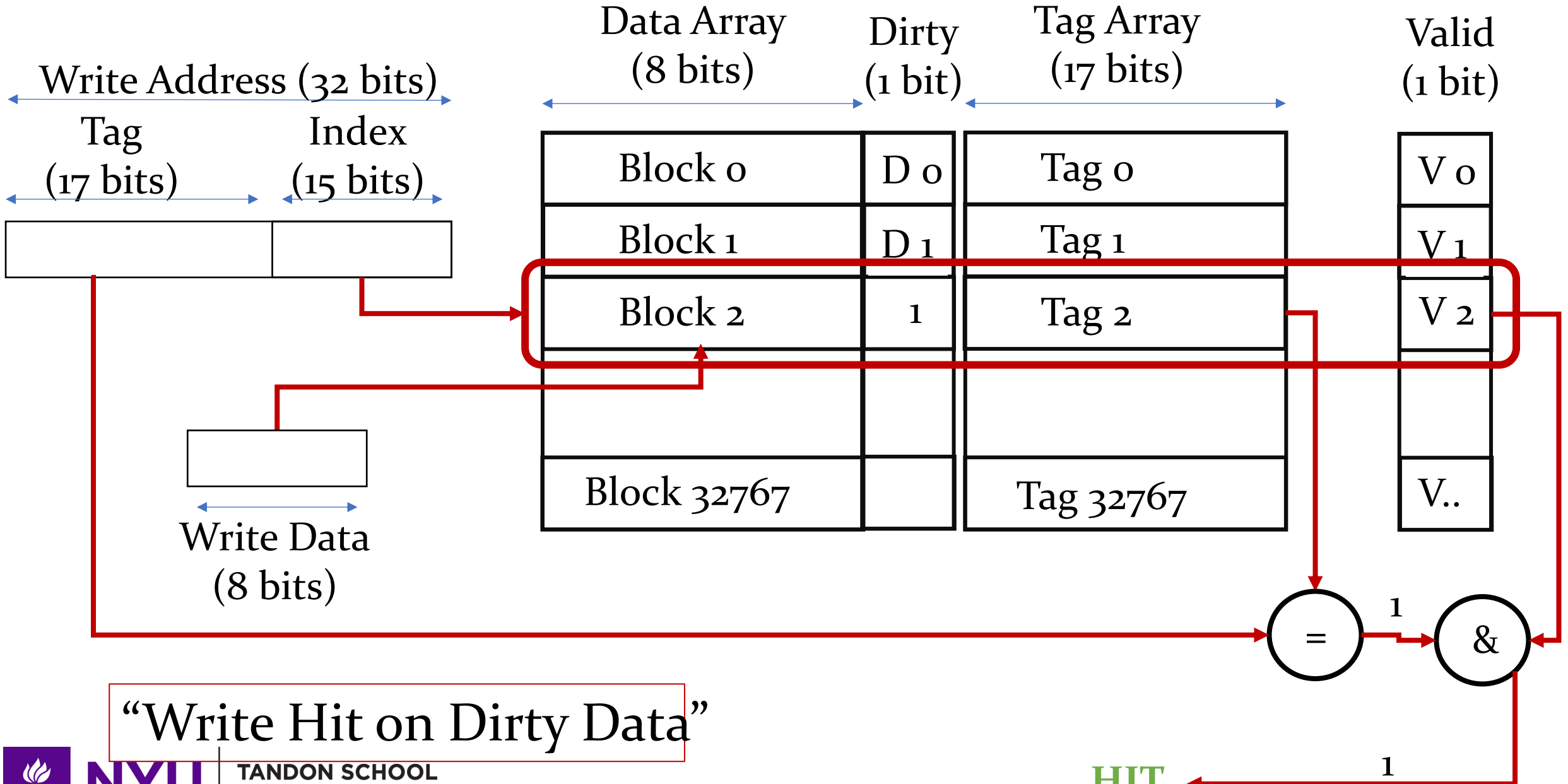
- Evictions take more time
- Data inconsistency between cache and lower levels

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



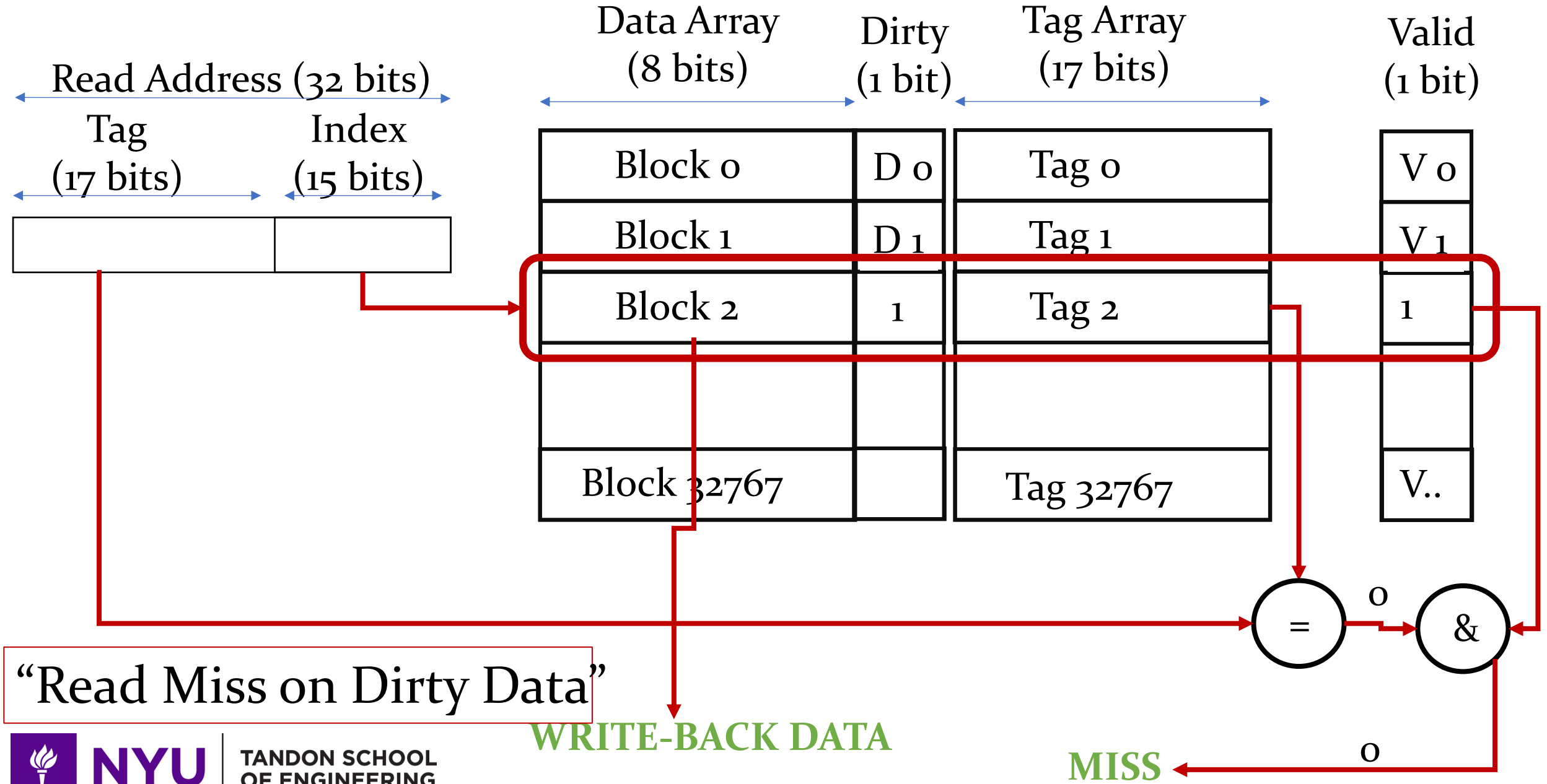
"Write Hit on Clean Data"

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



"Write Hit on Dirty Data"

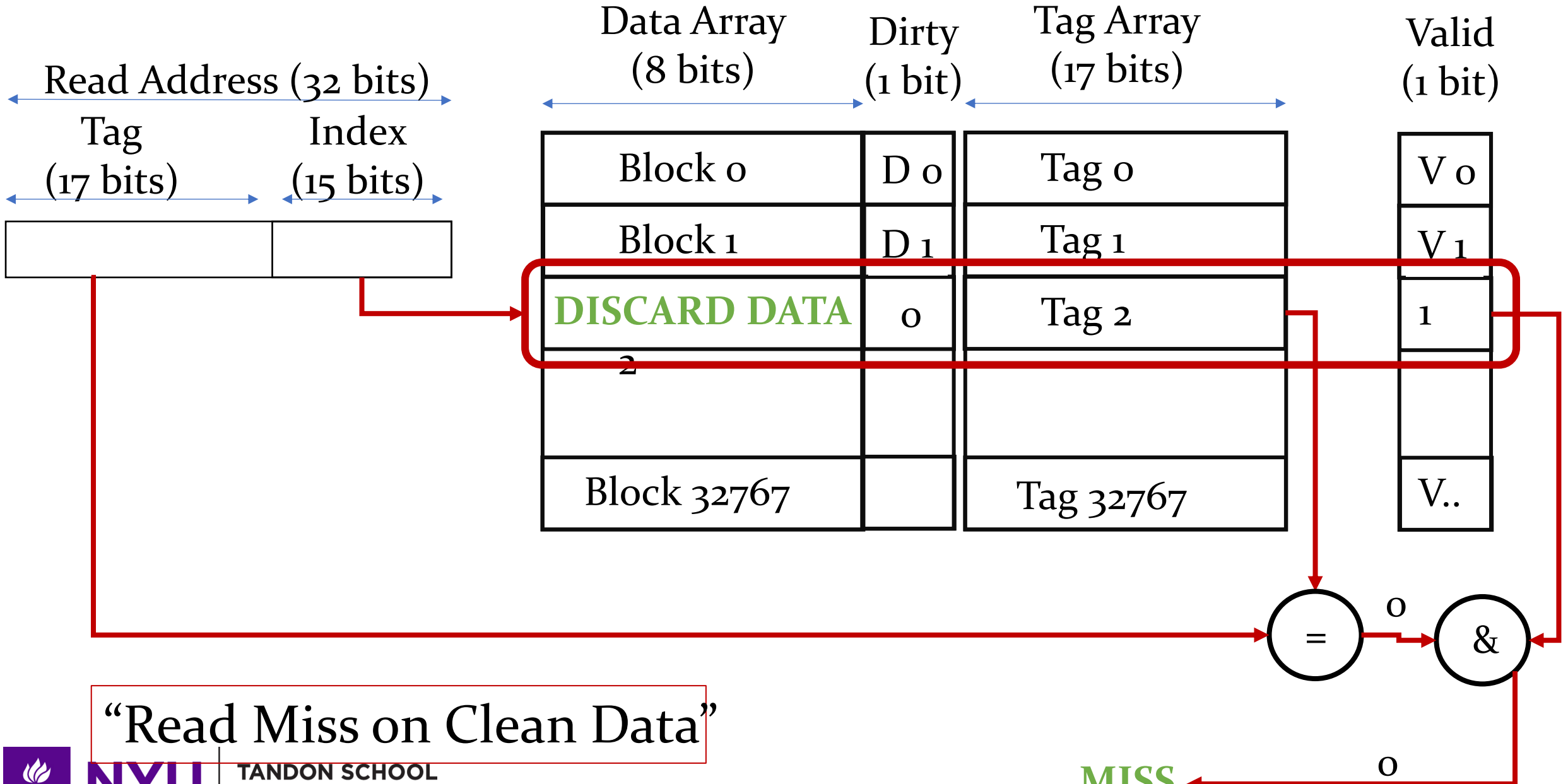
Write Back Cache (32 KB, Direct Mapped, 1 Byte Block)



"Read Miss on Dirty Data"

NYU TANDON SCHOOL OF ENGINEERING

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

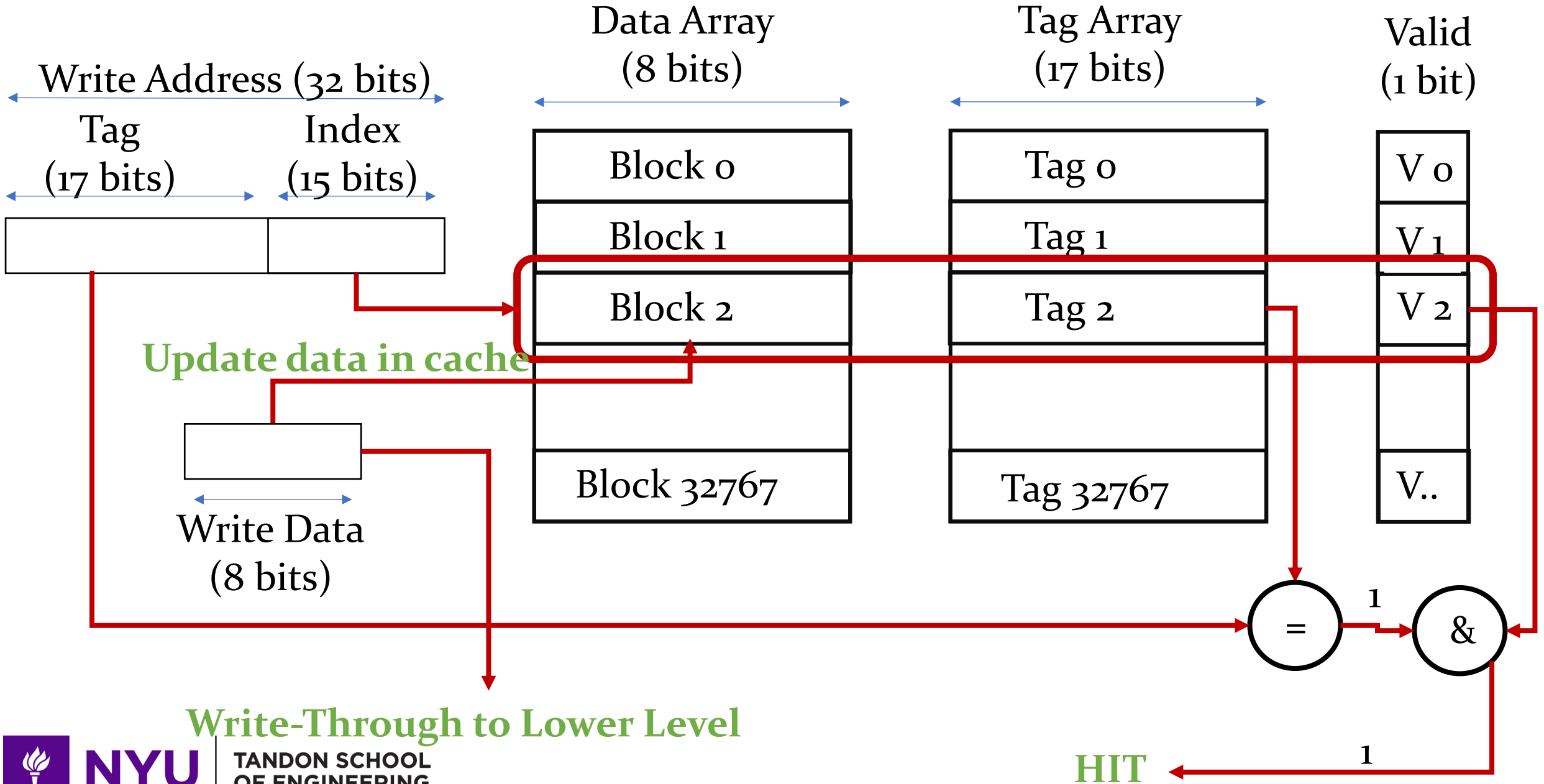
Pros

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

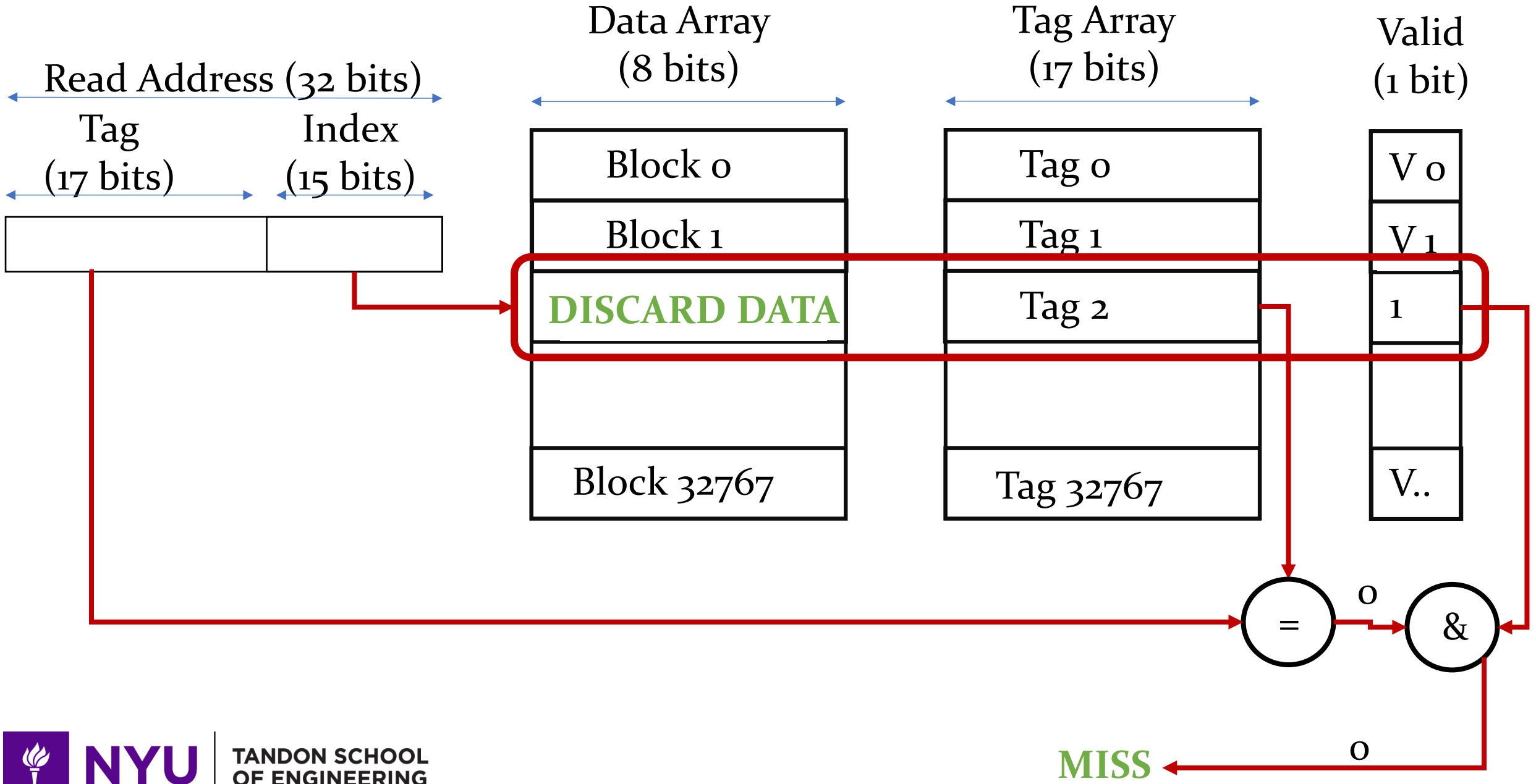
Cons

- 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)



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



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 data
- Standard write hit actions follow
- Good match for write-back caches

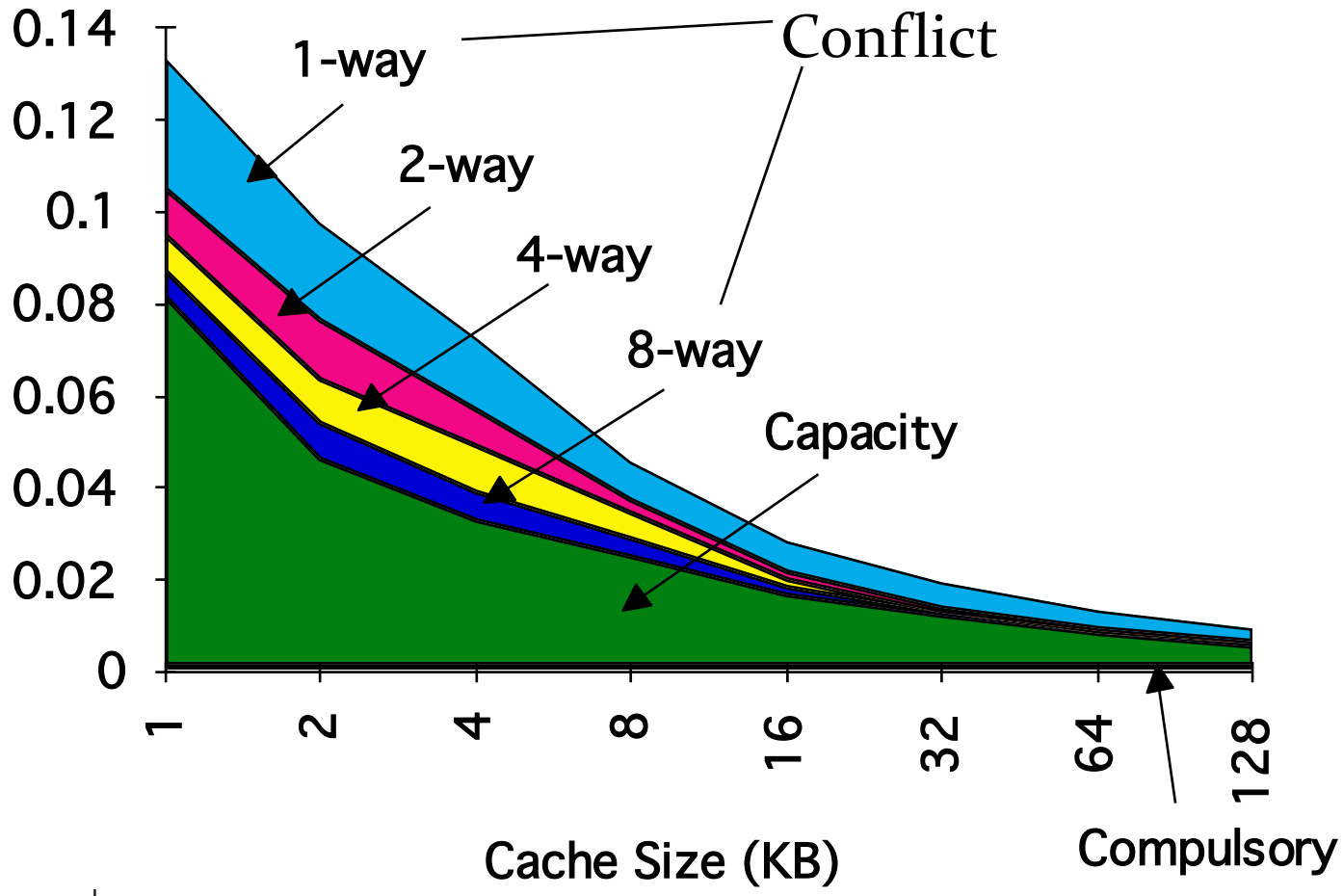
Write no allocate Policy

- 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

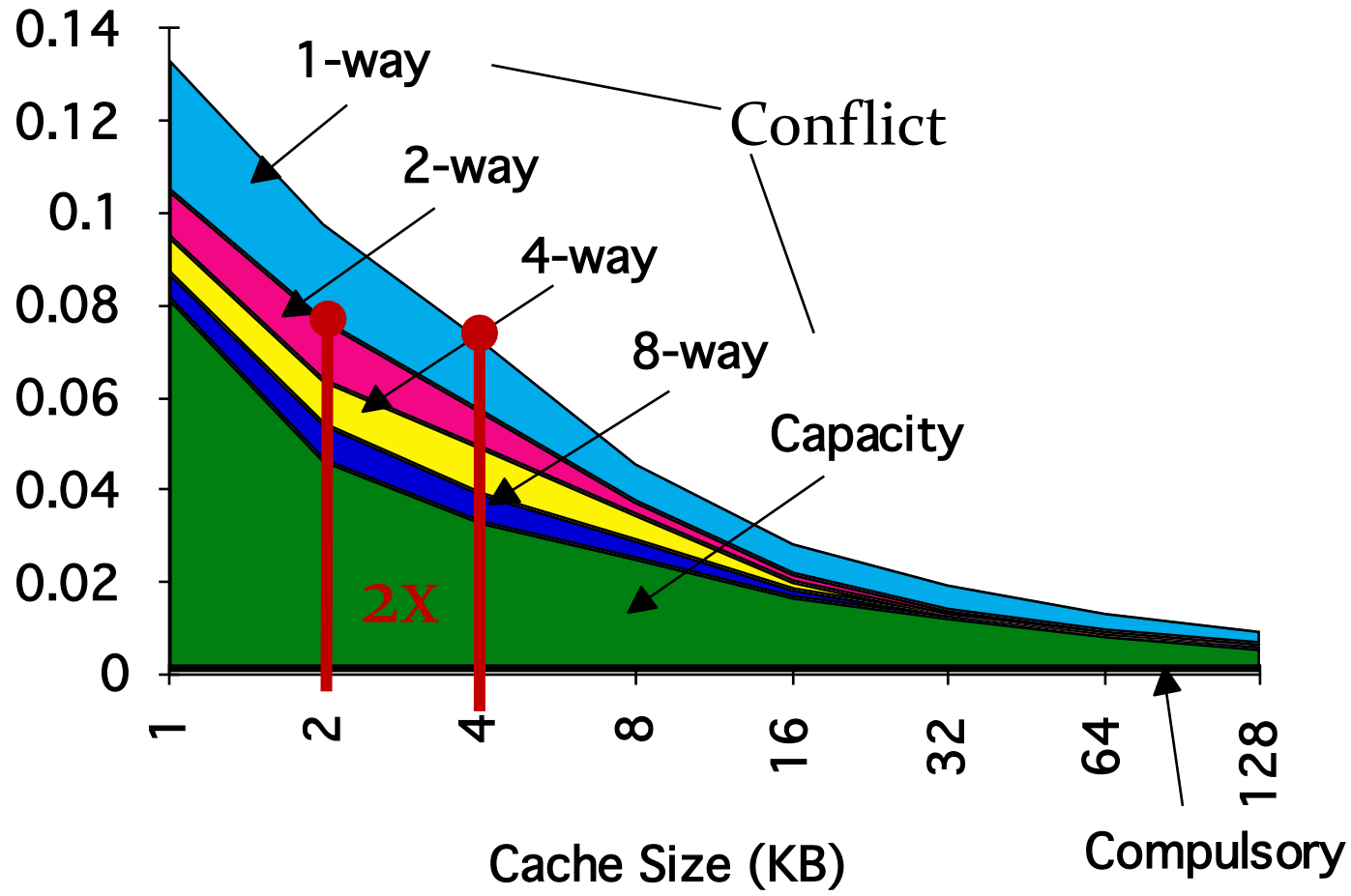
3Cs Absolute Miss Rate (SPEC92)

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

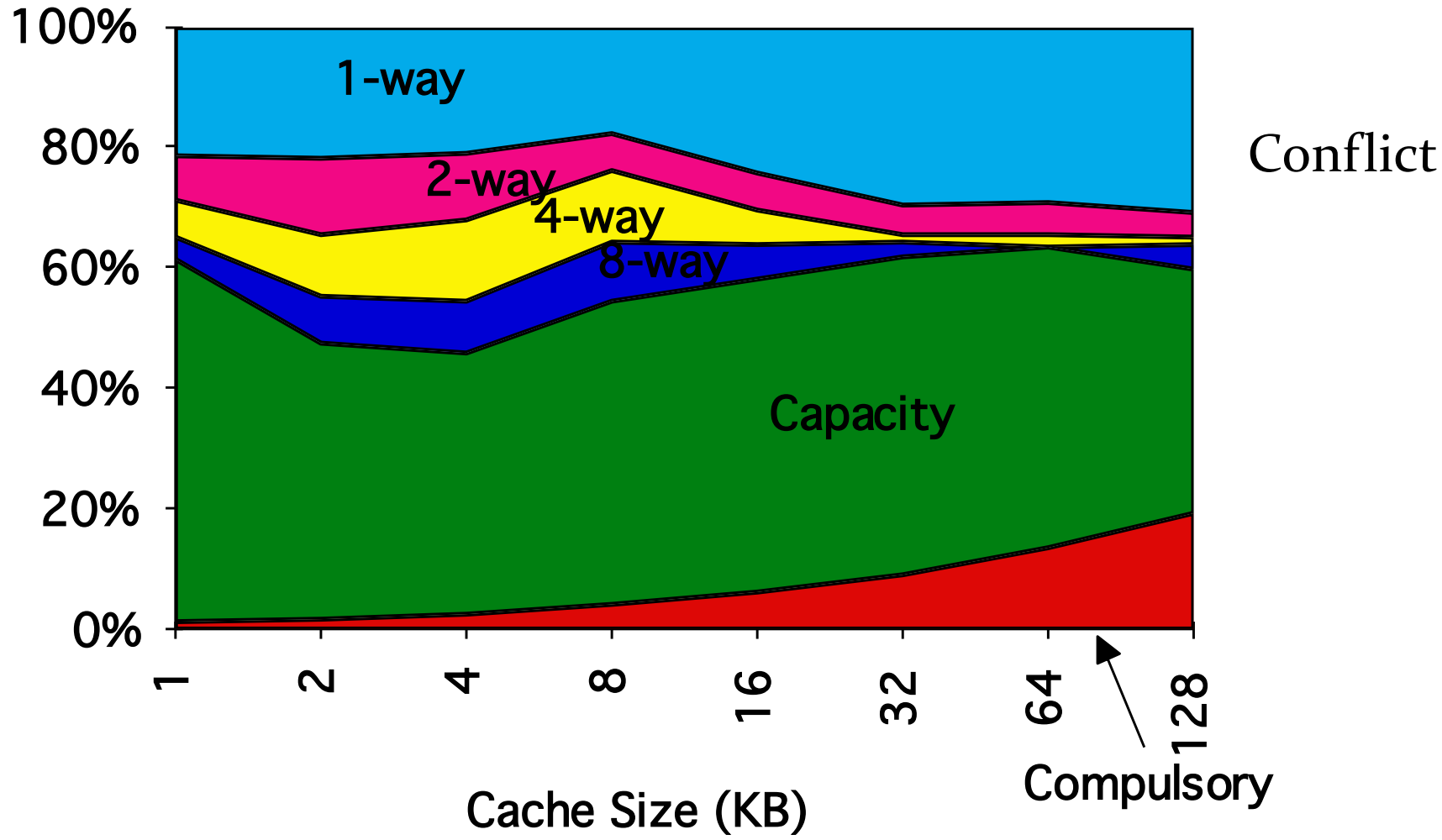


2:1 Cache Rule

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



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

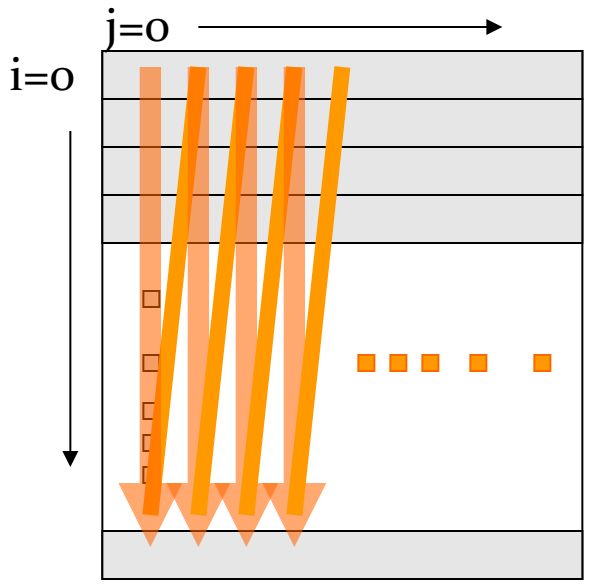
Examples:

- 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

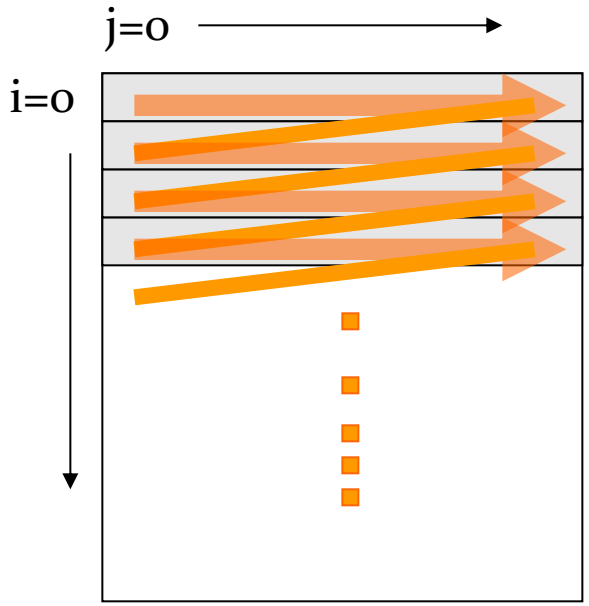
Row-major ordering

```
/* 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

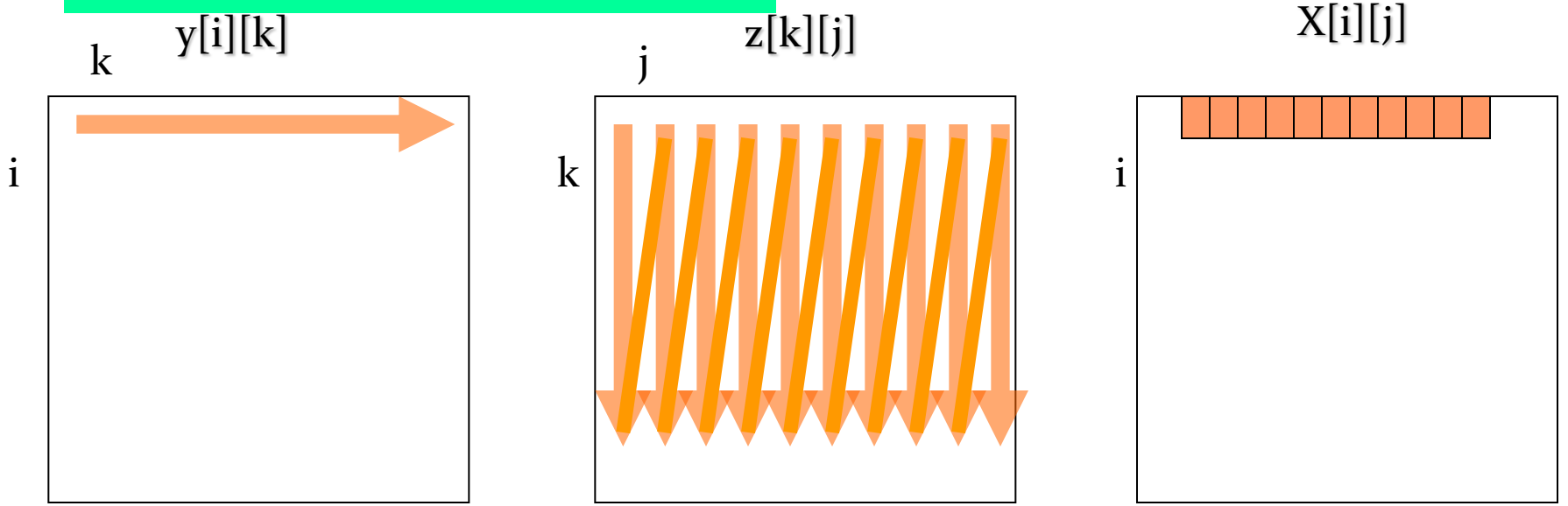
```
/* After */
for (i=0; i<5000; i++)
 for (j=0; j<100; j++)
   x[i][j] = 2*x[i][j]
```



Improved
cache efficiency

Loop Blocking

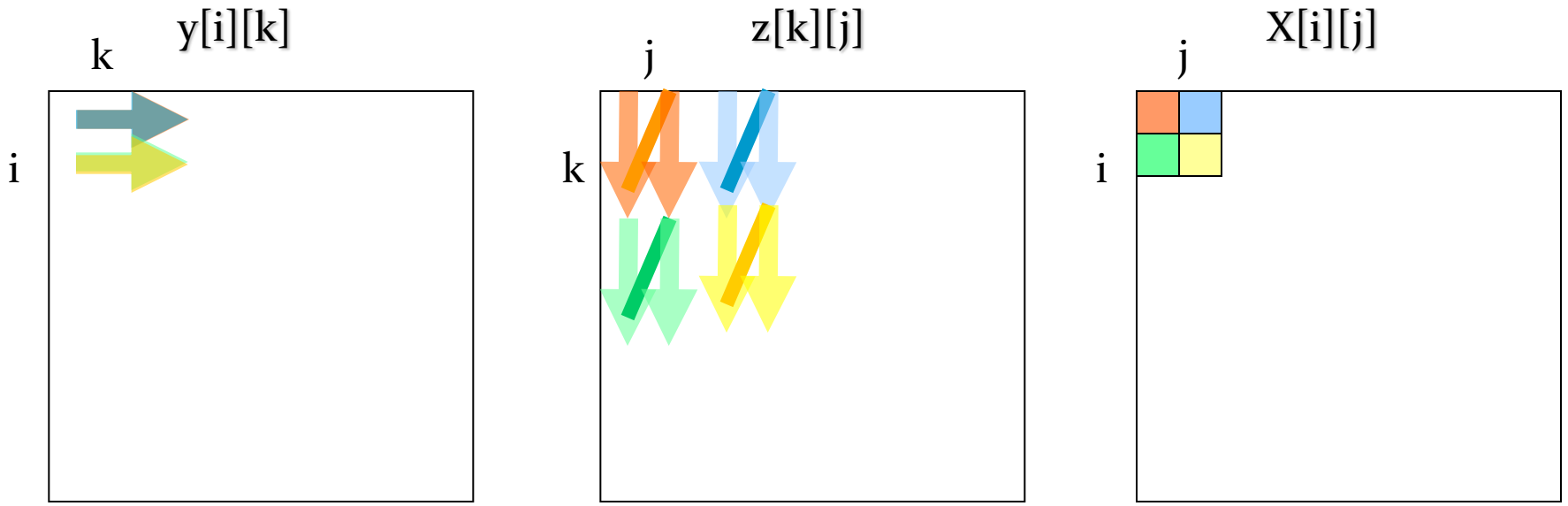
```
/* Before */
for (i=0; i<N; i++)
 for (j=0; j<N; j++) {
   r=0;
   for (k=0; k<N; k++)
      r += y[i][k]*z[k][j];
   x[i][j] = r;
 }
```



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

Victim Cache

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

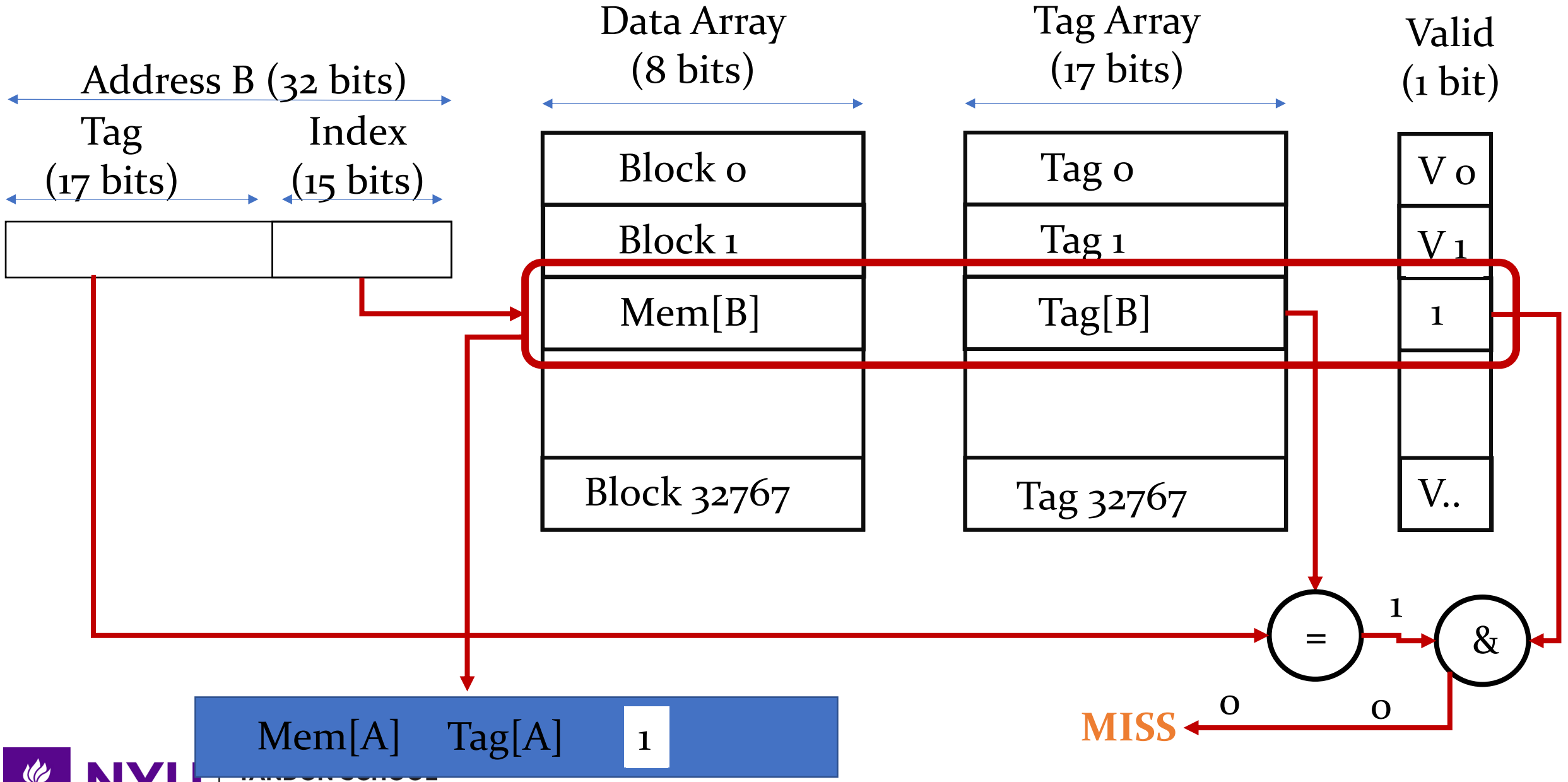
Jouppi [1990]:
4-entry victim cache removed
20% to 95% of conflicts for a
4 KB direct mapped data cache

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

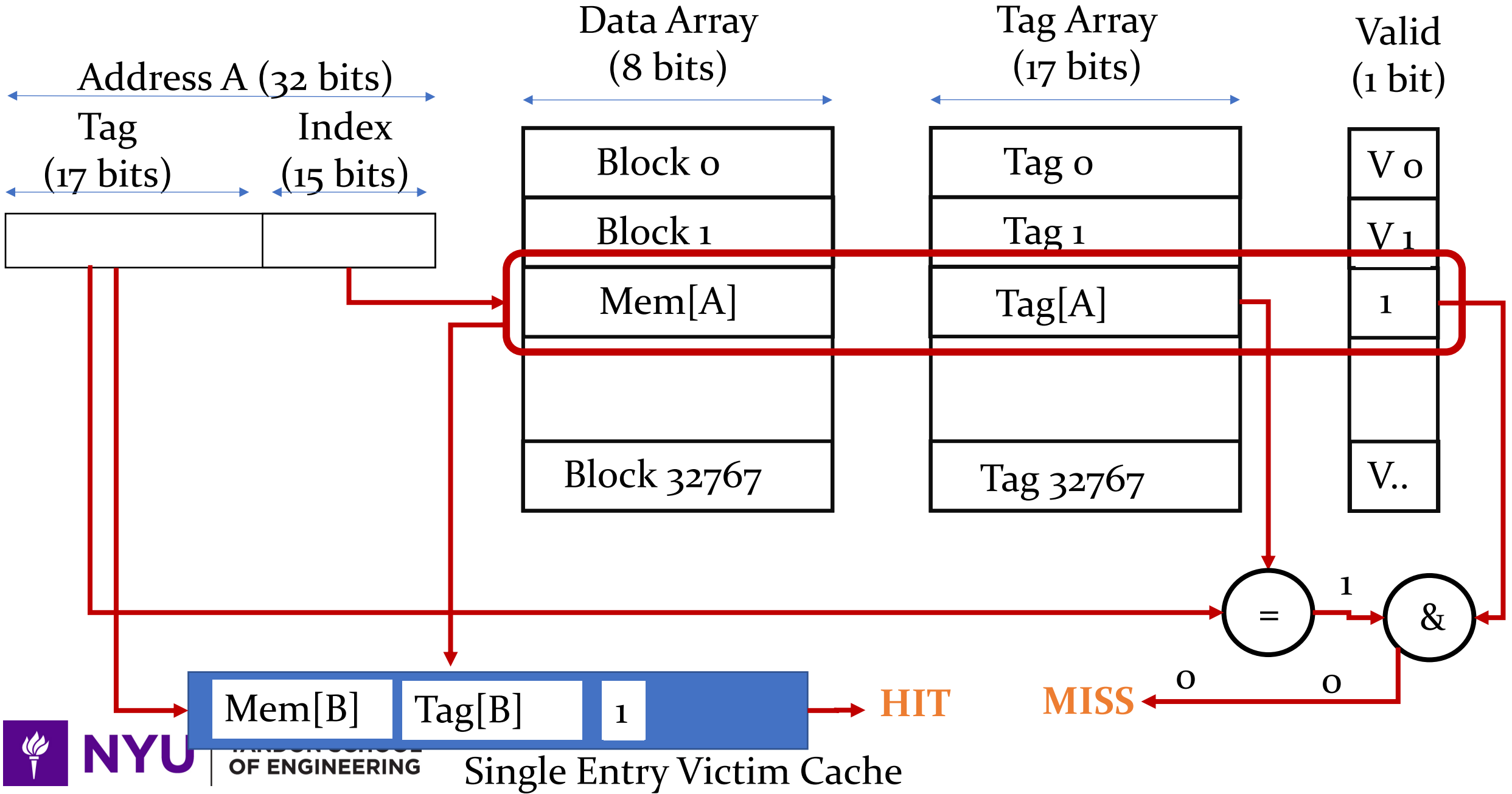
A victim cache is a small (4-8 entry) fully associative cache that holds blocks evicted due to conflict misses

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

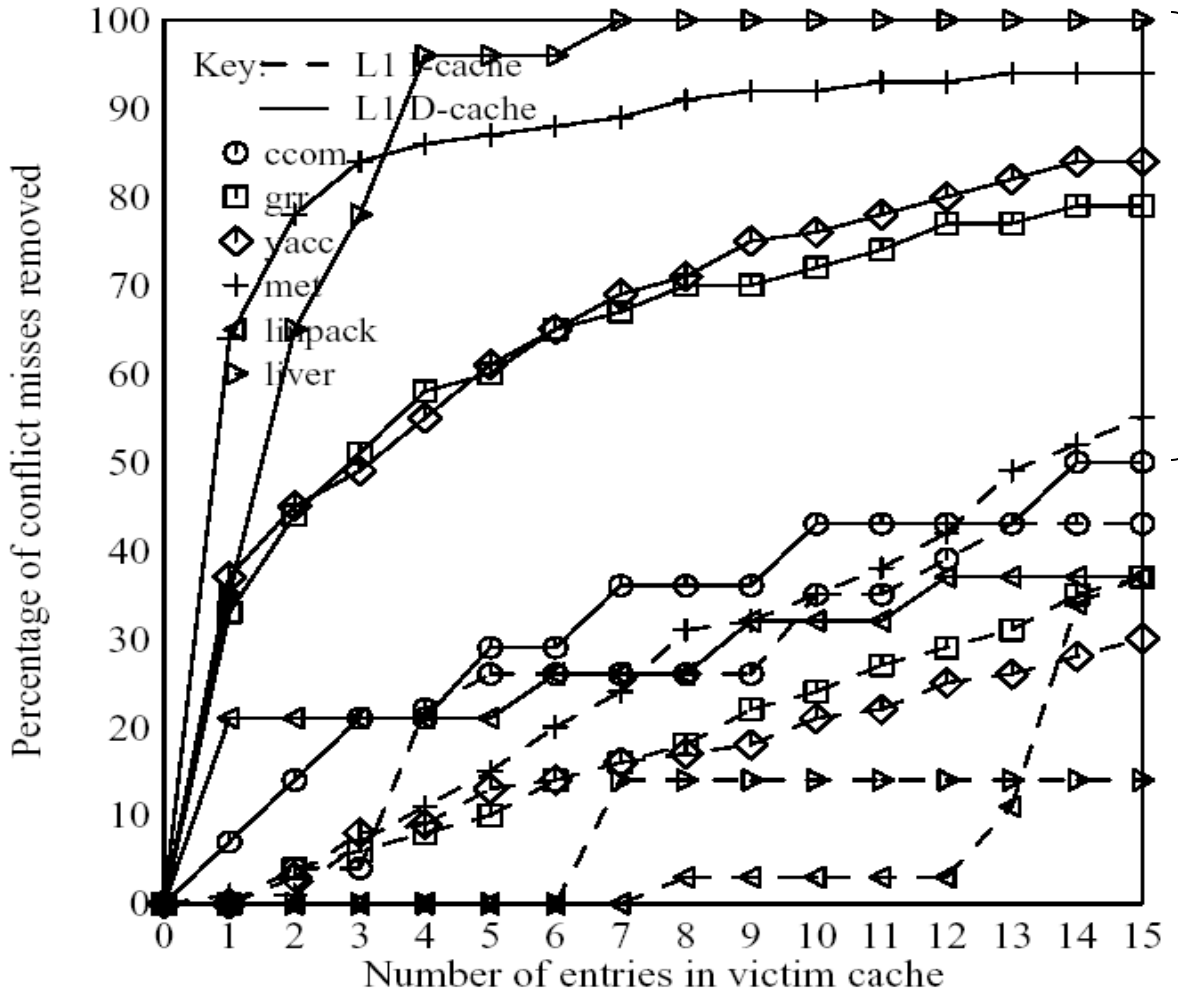
32 KB Direct Mapped Cache with 1 Byte Blocks + Victim Cache



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



% of Conflict Misses Removed

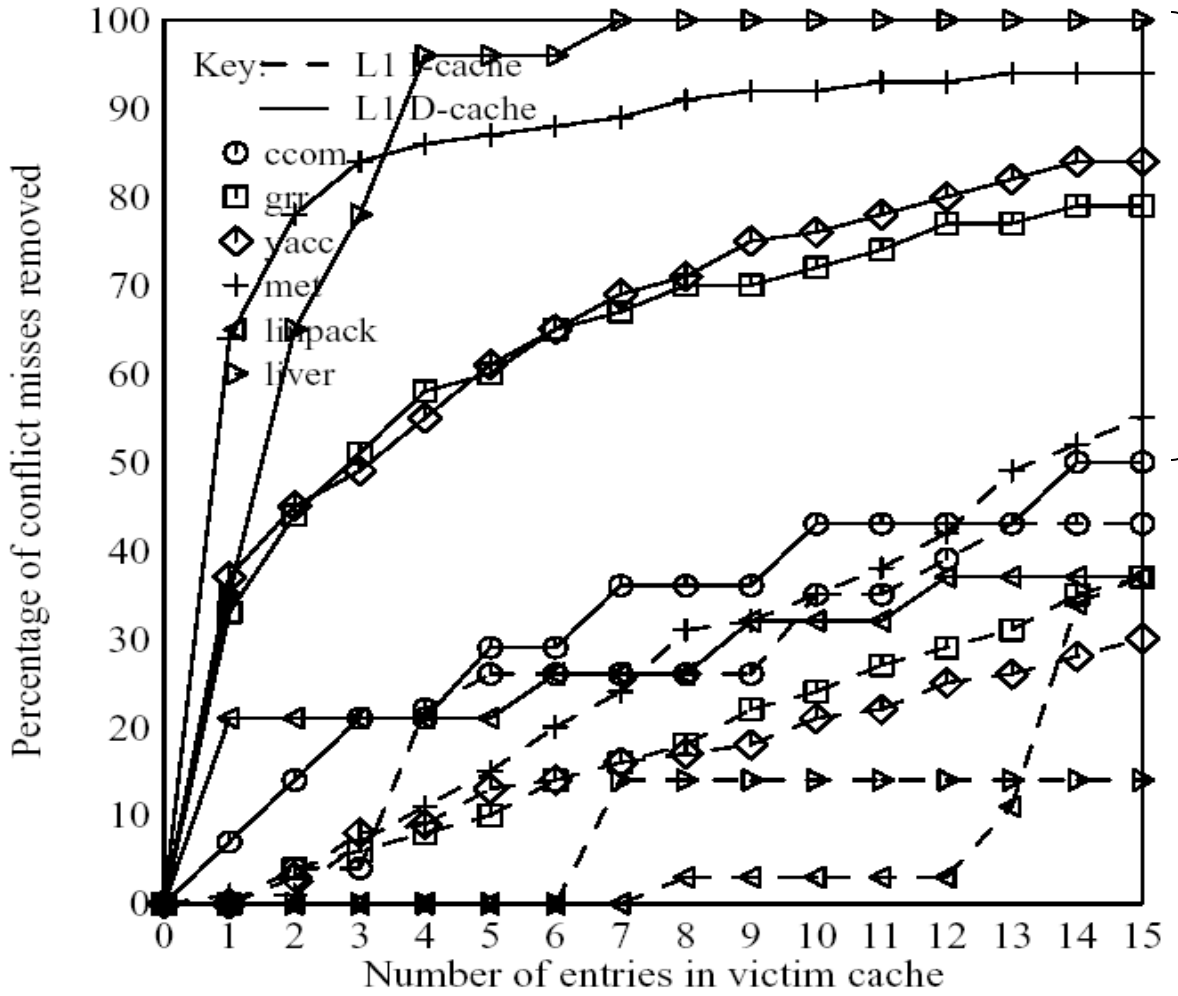


Dcache

Icache

Jouppi'90

% of Conflict Misses Removed



Dcache

Icache

Norm Jouppi…
What else did he do?

Jouppi'90

Built the TPU

AI & MACHINE LEARNING

Google supercharges machine learning tasks with TPU custom chip

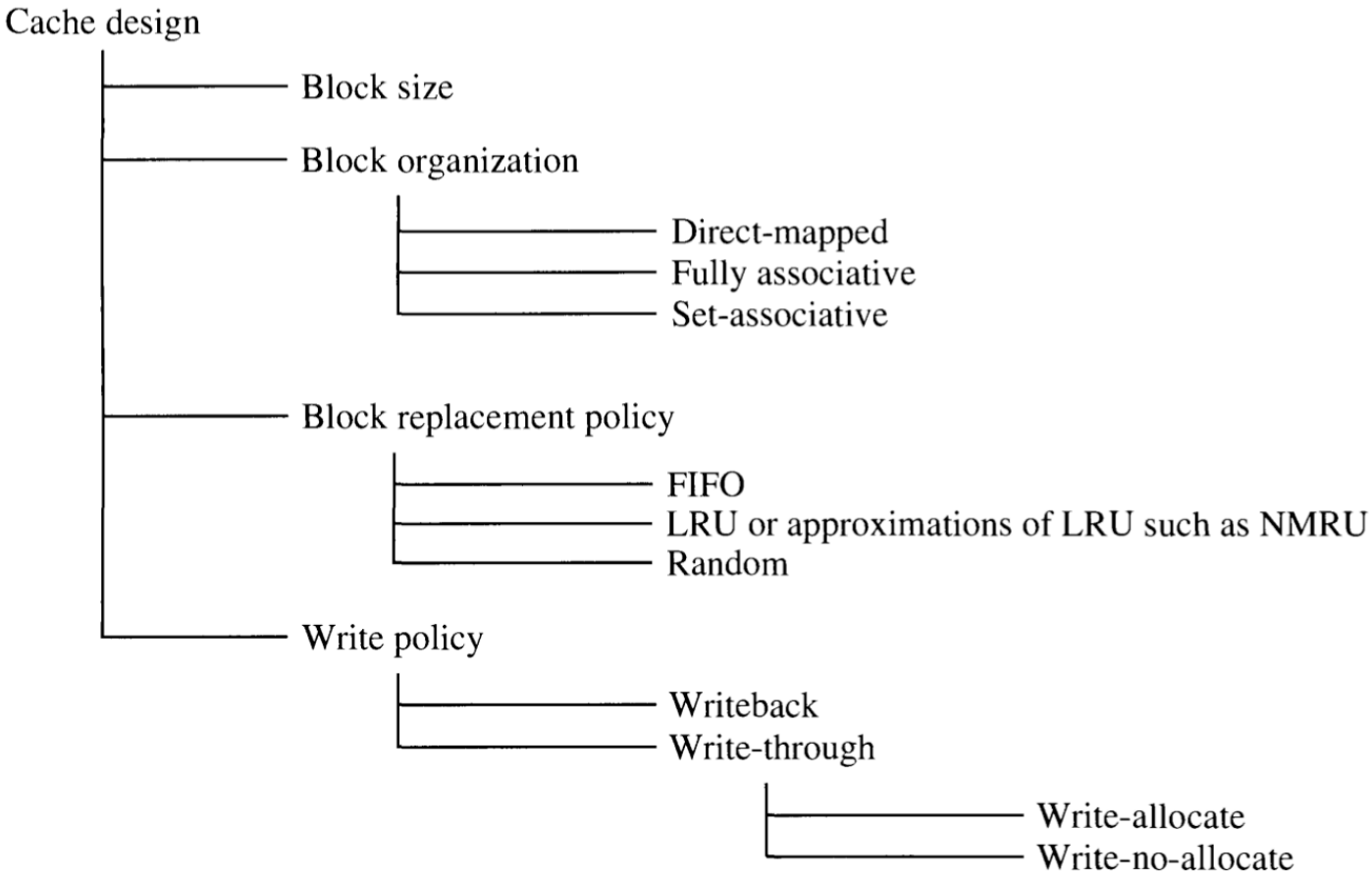
Norm Jouppi
Distinguished Hardware Engineer, Google

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.

Summary


Cache design
Block size
Block organization
Direct-mapped
Fully associative
Set-associative
Block replacement policy
FIFO
LRU or approximations of LRU such as NMRU
Random
Write policy
Writeback
Write-through
Write-allocate
Write-no-allocate

Please fill out course surveys!