Branch Prediction and Caches

Computer Architecture

ECE 6913

Brandon Reagen

NYU TANDON SCHOOL OF ENGINEERING

What we'll cover today

1) Logistics
2) Review and finish branch prediction
3) Introduce caches

Logistics

1) Homework 2 is released, solutions next week
2) Office hours Friday 11am-12pm
3) Course feedback (please fill out!)
4) Midterm Oct 22
    1) Will be a take-home test
    2) Should take ~2.5 hours
    3) Test will be released on Oct 22 at 11AM

        There will not be a lecture this day, use this time to complete the midterm

        To accommodate everyone, you'll have 25 hours to complete the test

        Issued: Oct 22, 11am

        Due: Oct 23, 12PM (noon)

Midterm material

1) MIPS ISA
2) MIPS datapath and pipelining
3) Hazards & forwarding
4) Performance metrics
5) Branch Prediction

We'll save caches for the final.

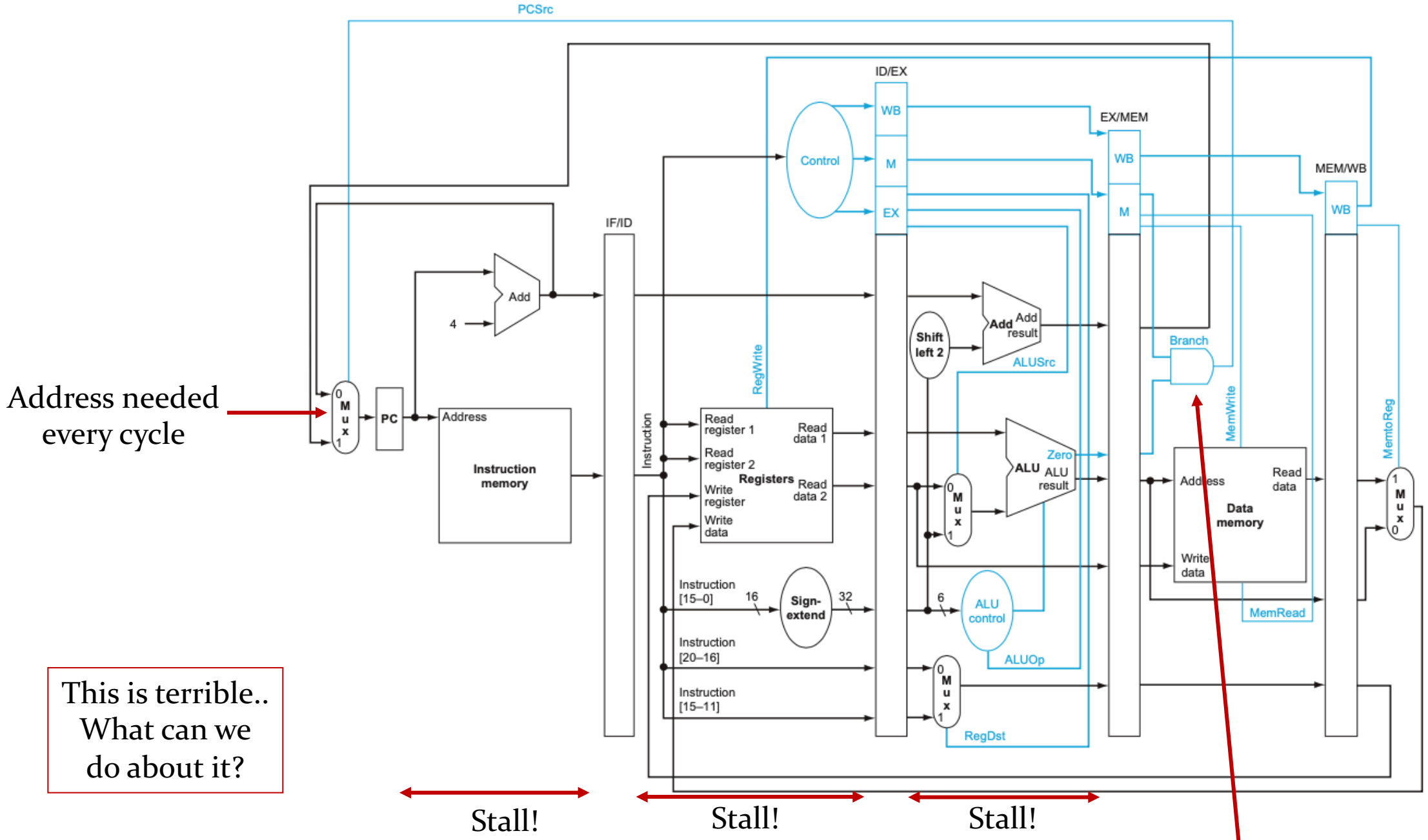
Hazards!

Hazard (formal) := “when the next instruction cannot execute in the following clock cycle”

Hazard (me) := “you need something you don’t have”

There are three of them..
Think about what goes into a program and how they execute

1) Structural : hardware resources : SOLVED!
2) Data : need result from prior computation : Forwarding!
3) Control : need to know where to go next => Branch prediction (later)



Address needed every cycle

This is terrible.. What can we do about it?

Stall!

Stall!

Stall!

Branch resolved

Solution 1: Assume not taken

- Assume not taken doesn't really change the design
  - Fetch instructions each cycle from PC+4
  - Resolve branches in MEM stage
- Pro:
  - What happens to CPI when branch resolves NOT taken?
- Con:
  - What happens to CPI when branch IS taken? (Common case..)
    - How do we implement this?
  - Flush: Discard instructions due to misprediction.
    - Insert NOPs
    - How and where?

Solution 2: Move branch resolution “up”

- Branches resolved in MEM stage… but why?
- We have all the information we need during ID!
  - Know it’s a branch, which registers to compare, value to update PC

- What does this do?
  - Reduces the branch delay
  - This implies that when we flush, we waste fewer cycles

- MIPS branch conditions intentionally kept simple
  - E.g., beq and bne only need to compare values
  - We can do this by adding small circuits to the ID stage

Challenges with early branching (1)

- Computing branch target address
  - First, we already have the info we need in IF/ID pipeline reg:
    PC and branch Immediate
  - Second, we can move the branch adder from EX stage to the ID stage

Challenges with early branching (2)

- Evaluating branch decision
  - New circuits late in the ID stage
    - After we read registers, have to compare values.
    - How can we implement comparators?
  - New bypass/forward logic!
    - Now forward data must also come into ID stage
      - Specifically, to the inputs of the equality unit
    - Can come from EX/MEM or MEM/WB pipeline regs
      - Requires new control logic
      - Cannot come from ID/EX regs, why?
    - Even with forwarding, we can stall. How?
      - If branch depends on previous ALU instruction: 1 cycle
      - If branch depends on previous MEM instruction: 2 cycles

Can only forward from the beginning of cycle

What happens when pipelines get "deeper" and "wider"?

- Assume a pipeline resolves branches in stage 20 (not 2) and fetches 5 inst per cycle
- Also, 20% (1 in 5) instructions are branches, uniform distribution
- How long does it take to fetch 500 instructions?

  100% accuracy:
  - 100 cycles, no flushes => No wasted work!

  99% accuracy:
  - 100 (correct) + 20 (penalty) * 1 (wrong frequency*branches) = 120 cycles
  - 20% more cycles..

  95% accuracy
  - 100 (correct) + 20 (penalty) * 5 (wrong frequency * num branches) = 200 cycles
  - 100% extra instructions fetched.

Dynamic Hardware Branch Prediction

- Need to do better than always fetching not taken instructions
  - Performs poorly (think about loops)
- Branch prediction
  - **Guess** which direction to go!
- Dynamic
  - Use results of recently taken branches while program executes
- Hardware
  - Implement with special circuits for speed.
  - Would branch prediction be part of architecture or microarchitecture?
  - Could we do it in software?

Why are Branches Predictable?

```
for (i=0; i<100; i++) {

        ….
}
```

```
addi   r10, r0, 100
add    r1,   r0, r0

L1:
… …
… …
addi   r1, r1,   1
bne    r1, r10, L1
… …
```

```
if (aa==2)
        aa = 0;
if (bb==2)
        bb = 0;
if (aa!=bb)
        ….
```

```
addi  r2,   r0, 2
bne   r10, r2, L_bb
xor   r10, r10, r10

L_bb:
bne   r11, r2, L_xx
xor   r11, r11, r11

L_xx:
beq   r10, r11, L_exit
…
Lexit:
```

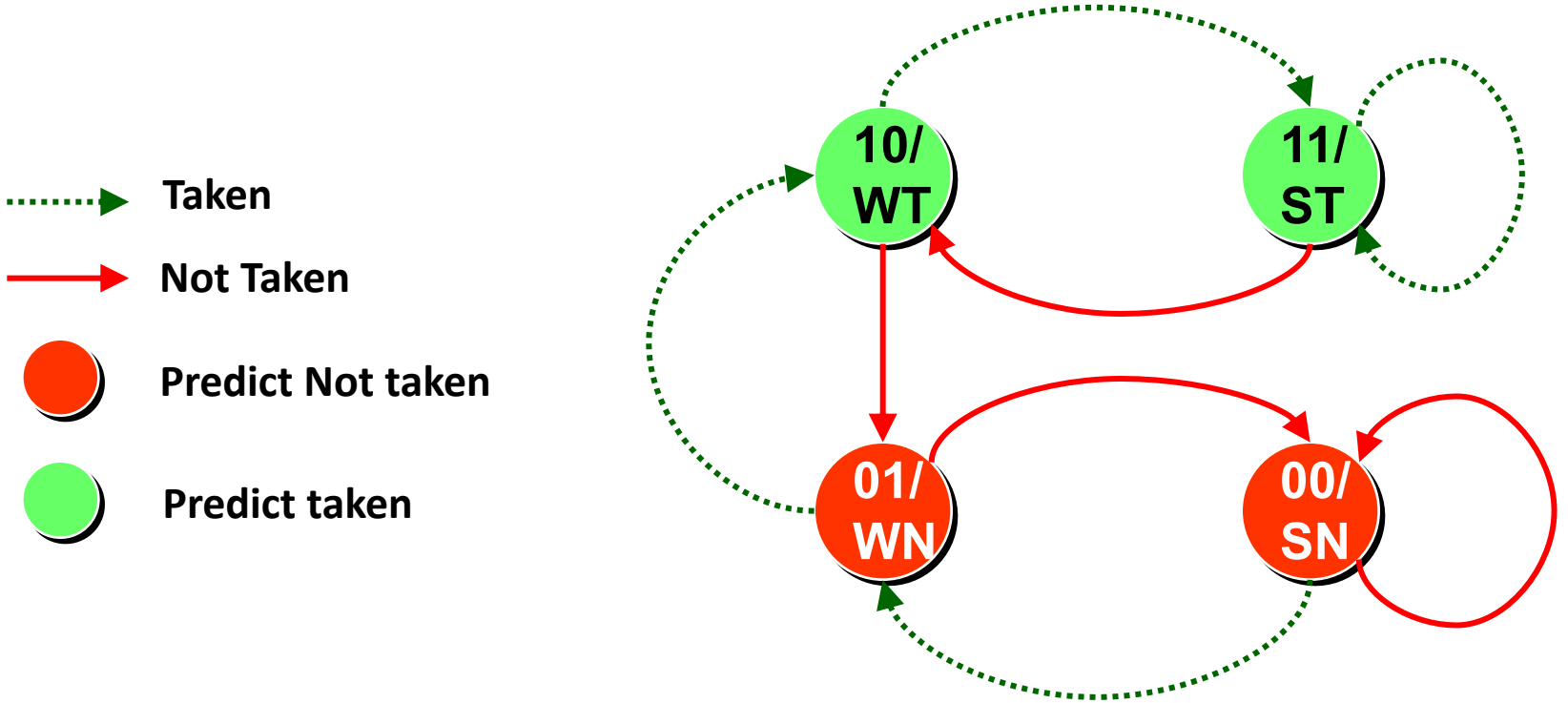
Branch predictors have two jobs

This is what we'll focus on today.

- In order to perform task, predictors must
  - Speculate whether the branch is to be taken or not-taken
  - Calculate the target address
    - Target address: Where to go when branch predicted taken
    - What about unconditional jumps?

- Past predicts the future
  - To make informed decisions, we'll track previous branch directions
    - As we'll see, this sounds simple but gets quite complex

2-bit Saturating Up/Down Counter Predictor



ST: Strongly Taken
WT: Weakly Taken
WN: Weakly Not Taken
SN: Strongly Not Taken

MSB tells us direction (taken/not taken)
LSB servers as hysteresis or inertia

The “Smith Predictor” (1981)

General solution:
- m determines # entries (FSMs)
- k is the number of bits/entry

Total size of the predictor is: k2^m

Rather than just using single 2b counter for everything, create a table of counters/FSMs

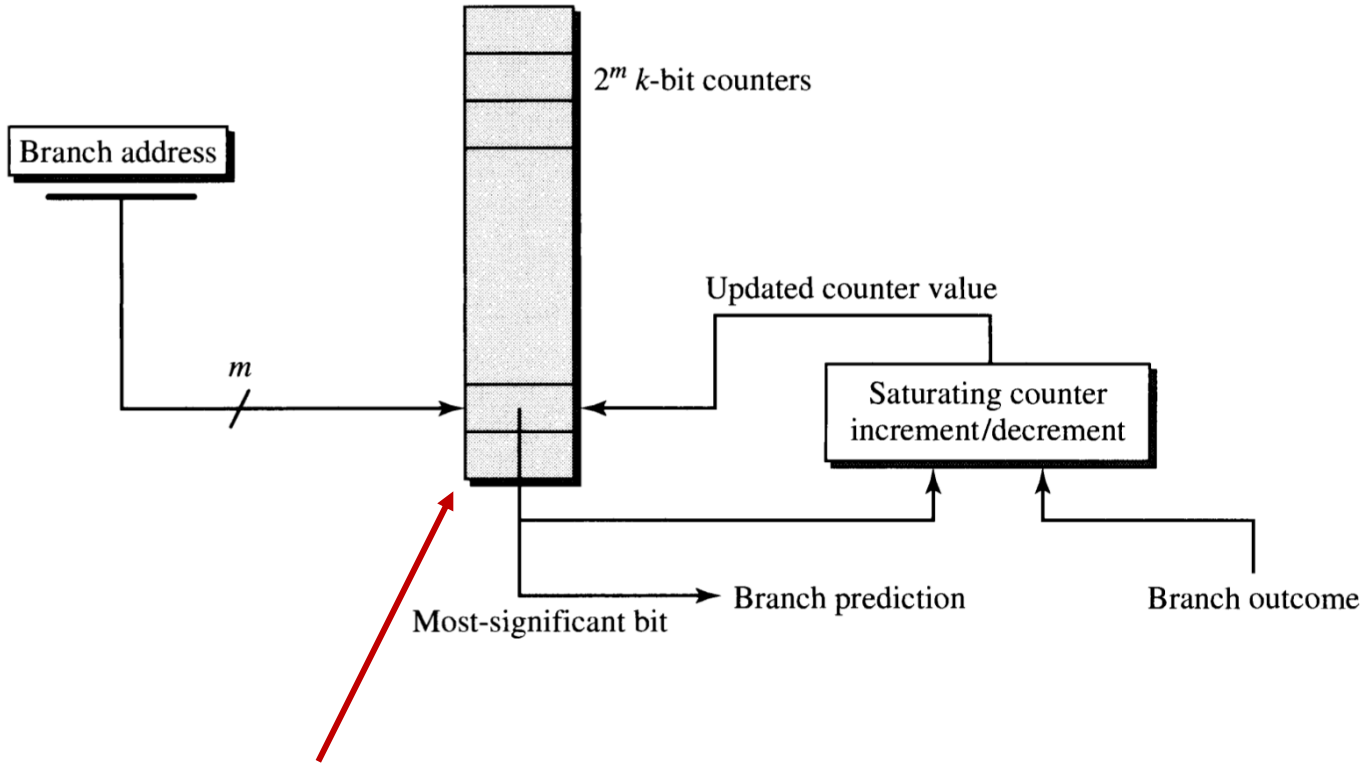
Index this table using the branch address (PC)

Option 1: Use entire branch address
Too expensive!

Option 2: Use 1 entry/branch
Too expensive!

Option 3: Hash and hope collisions aren’t too bad

Option 3 is used today. Some uses actual hash functions some take LSBs



Called the Pattern History Table (PHT)

Two –Level Prediction Tables

More advanced solution: Two–Level adaptive branch prediction

"Two-level" (Yeh and Patt, 91, 92, 93)
"Correlation branch prediction" (Pan, 92)

Key idea: Different branches tend to impact each other. We can make better predictions by using information from other recent branches (both locally and globally).

Smith's predictor – Captures best branch decision

Two-level predictor – Also captures recent decision history

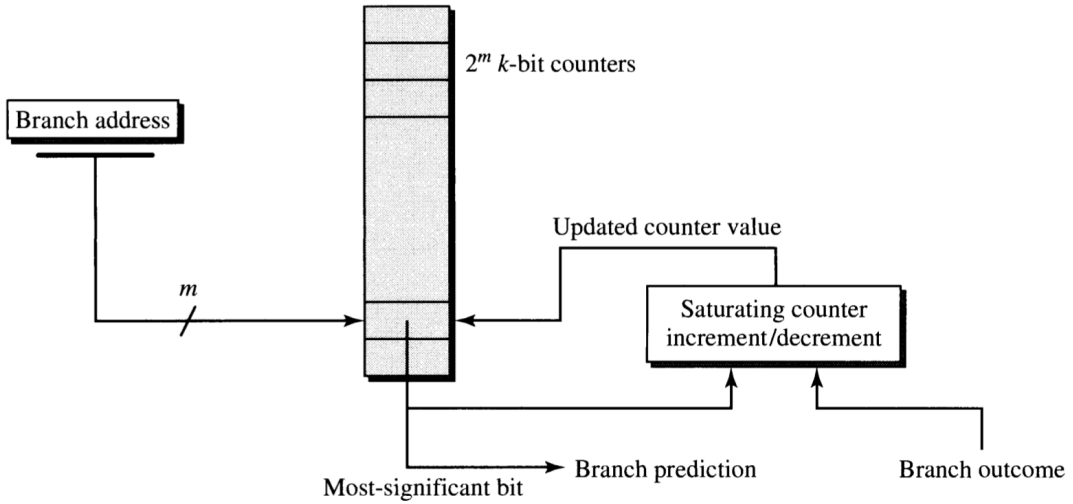
Key idea: Track global branch decisions

Branch history register (BHR)

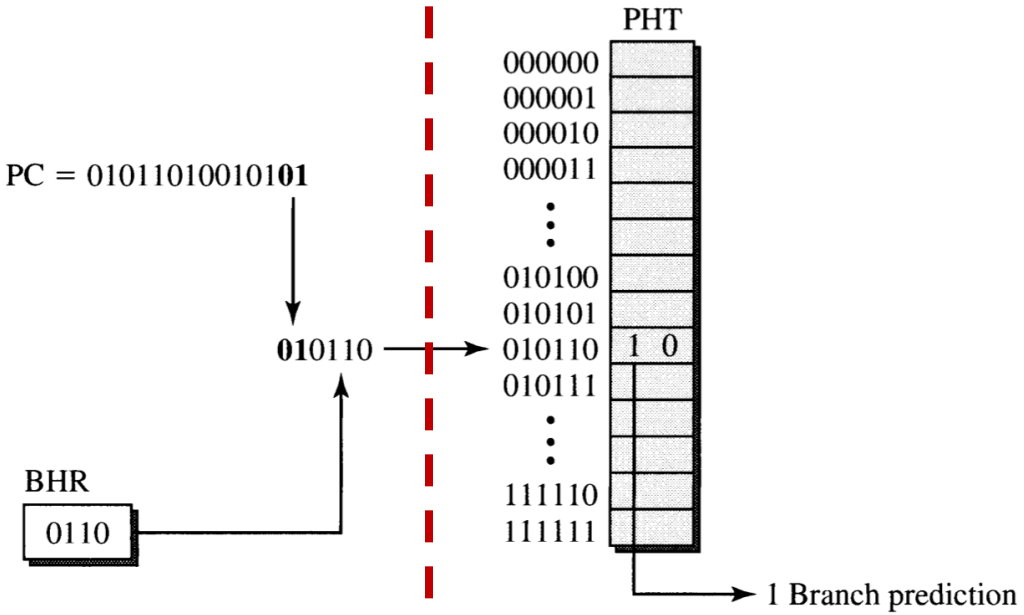
- Use a single register to keep track of all recent branch outcomes
- Typically use a shift register
  - When branch resolves, shift in 1 for taken, 0 for not taken
- Use the BHR or BHR+PC to index PHT
  - Remember, PHT is just a bunch of FSMs
  - The BHR provides a different way of picking PHT entry

Example of a two-level predictor

Smith's



Two-level predictor



Level 1
Track recent outcomes in BHR
Uses h bits, in this case 4
Implemented as Queue (LIFO) right->left

Level 2
Pattern history table (just like before)
Each entry is 2b saturating counter
Still updates on each branch outcome

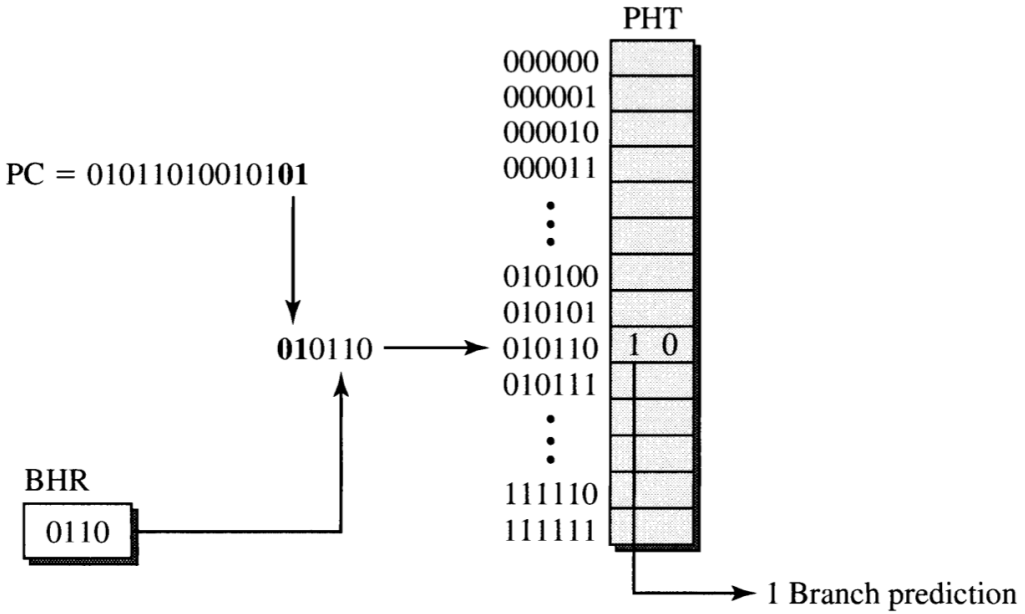
Indexing tradeoffs

How do we decide the PHT address?

More PC bits implies fewer collisions
Collisions: different PC, same FSM
More BHR bits implies more correlation

Architects job is to balance these competing tradeoffs

What does Smith's predictor favor?



Intuition

```
If (aa==2)
        aa=0
If (bb==2)
        bb=0
If (aa!=bb){
        <do something>
}
```

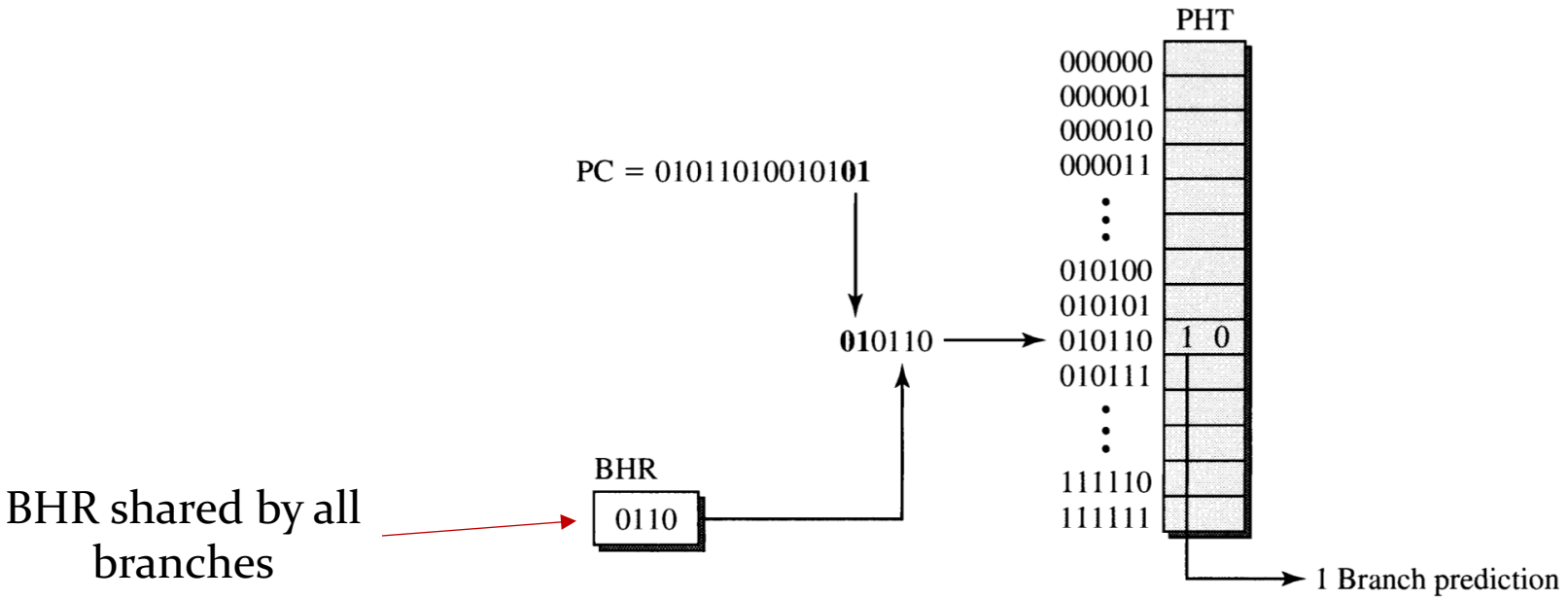
```
X = 0
If (someCondition)
        x=3
If (someOtherCondition)
        y+=19
If (x <= 0){
        <do something>
}
```

High correlation

Low correlation

BHR lets us capture these correlations!

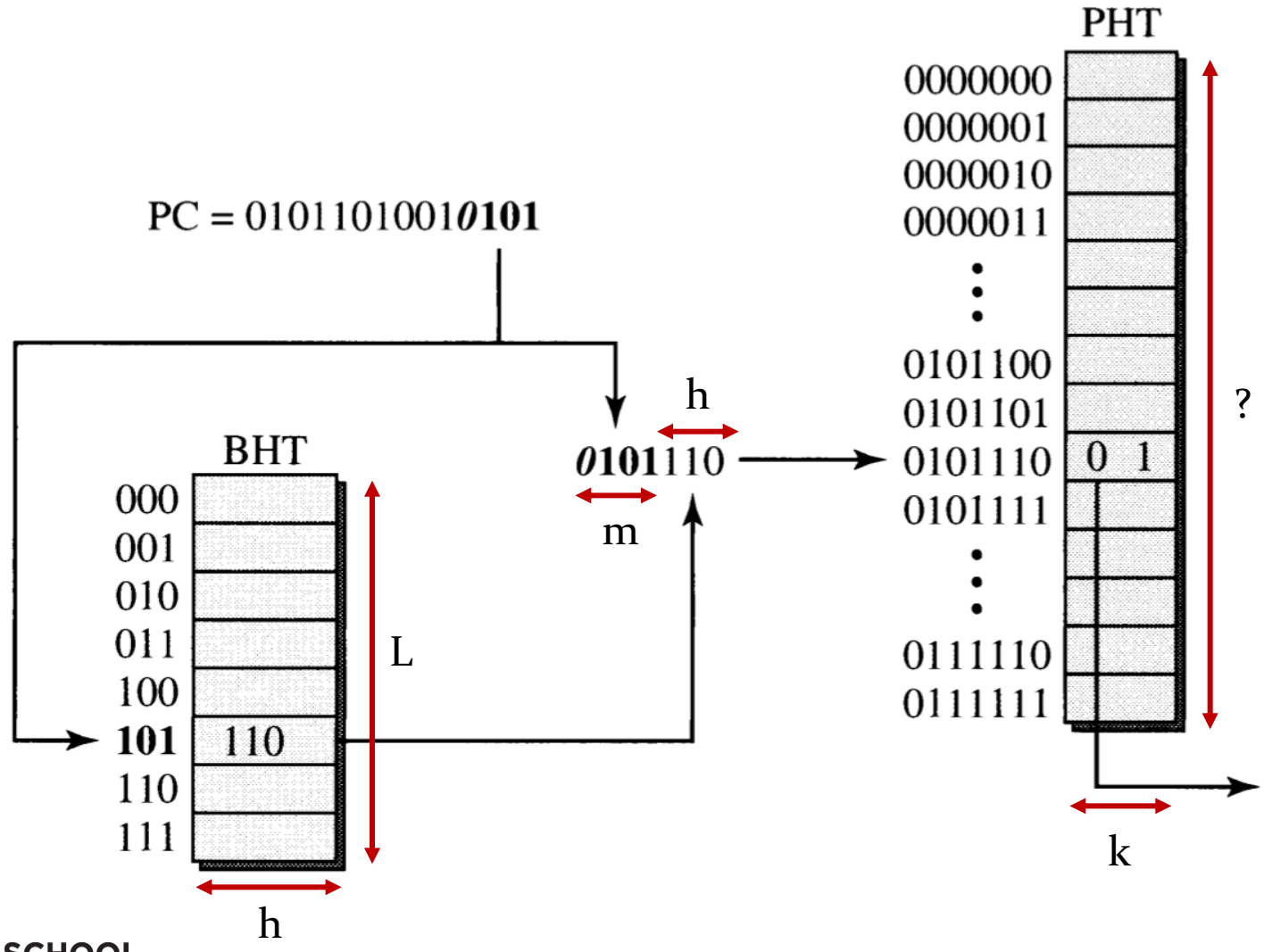
“Global” two-level predictor



“Local-history” two-level predictor

- Global BHR tracks history of all recent branches
- Local tracks outcomes of last several encounters of current branch
  - Generalization of the Global two-level predictor
  - Extend the single BHR to an array of BHRs
  - Called the Branch History Table (BHT)
  - Index the BHT using the LSBs of the PC
    - Each BHR tracks the branch history for that branch
    - Think of it as:
      - BHT index = look up the history pattern for this branch (PC)
      - BHT entry = the last few times this branch occurred, here’s what happened
      - PHT index = Given I saw that pattern for that branch, what should I do?
      - PHT entry = I’ll know what to do looking at the MSB of the 2b counter!

Local-History Two-Level Predictor



How big is this table?

Lh + k2^(m+h)

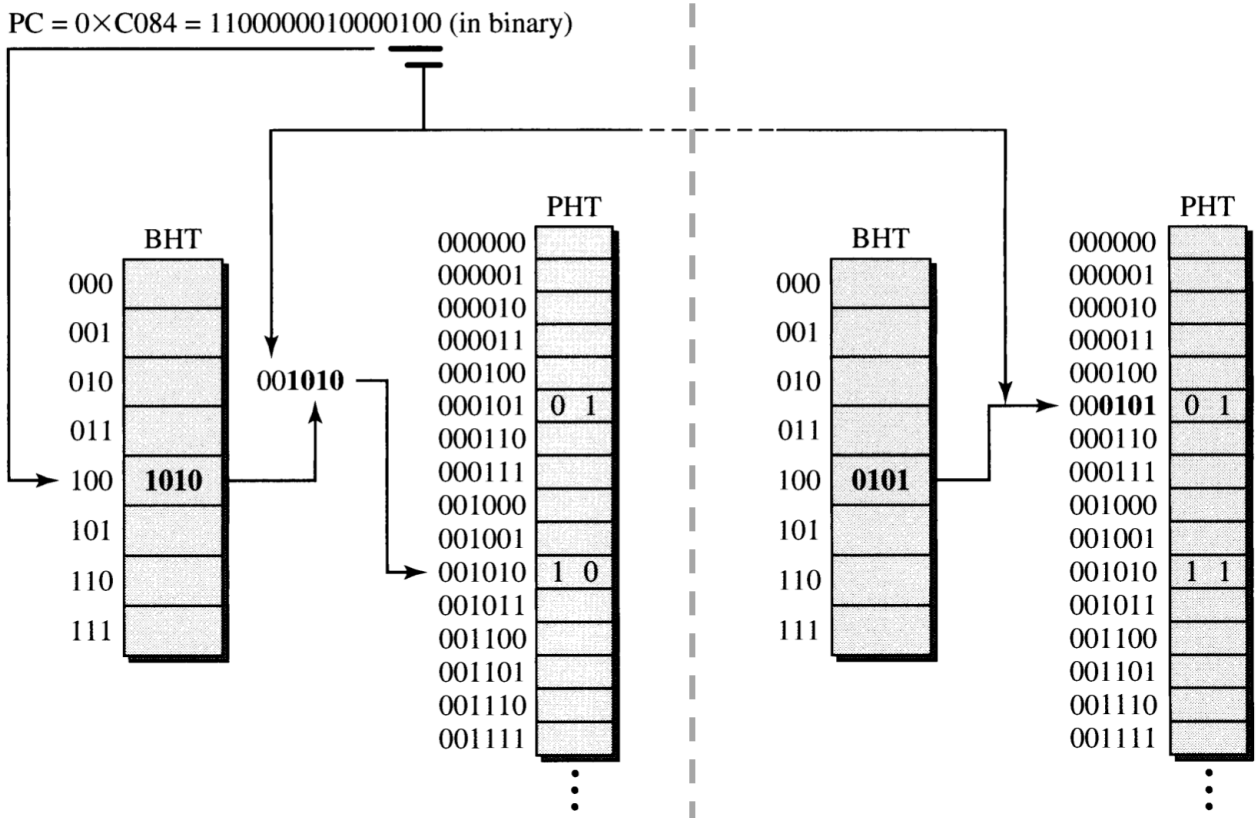
8*3 + 2*2^(7)
= 24+256
=280bits

Example 1: T, N, T, N, …

- Simple pattern, is it simple to predict?
- What happens with 1b predictor initialized to not taken?
  - 0% accuracy, wrong every time!
- What about 2b saturating predictor initialized to Weakly T?
  - 50%
  - What if it's initialized to Strongly NT?

- What about a Locally-History Two-level predictor?

Example 1: Local-History 2-level predictor T, NT, T, NT



First
Guess: taken
Result: taken

Second
Guess: not taken
Result: not taken

What's going on here?

1) Always point to same BHR
2) PHT changes based on local branch history (key!)
3) Different FSMs tell us what to do depending on what we last saw

Intuition: Use both PC and pattern to make prediction.

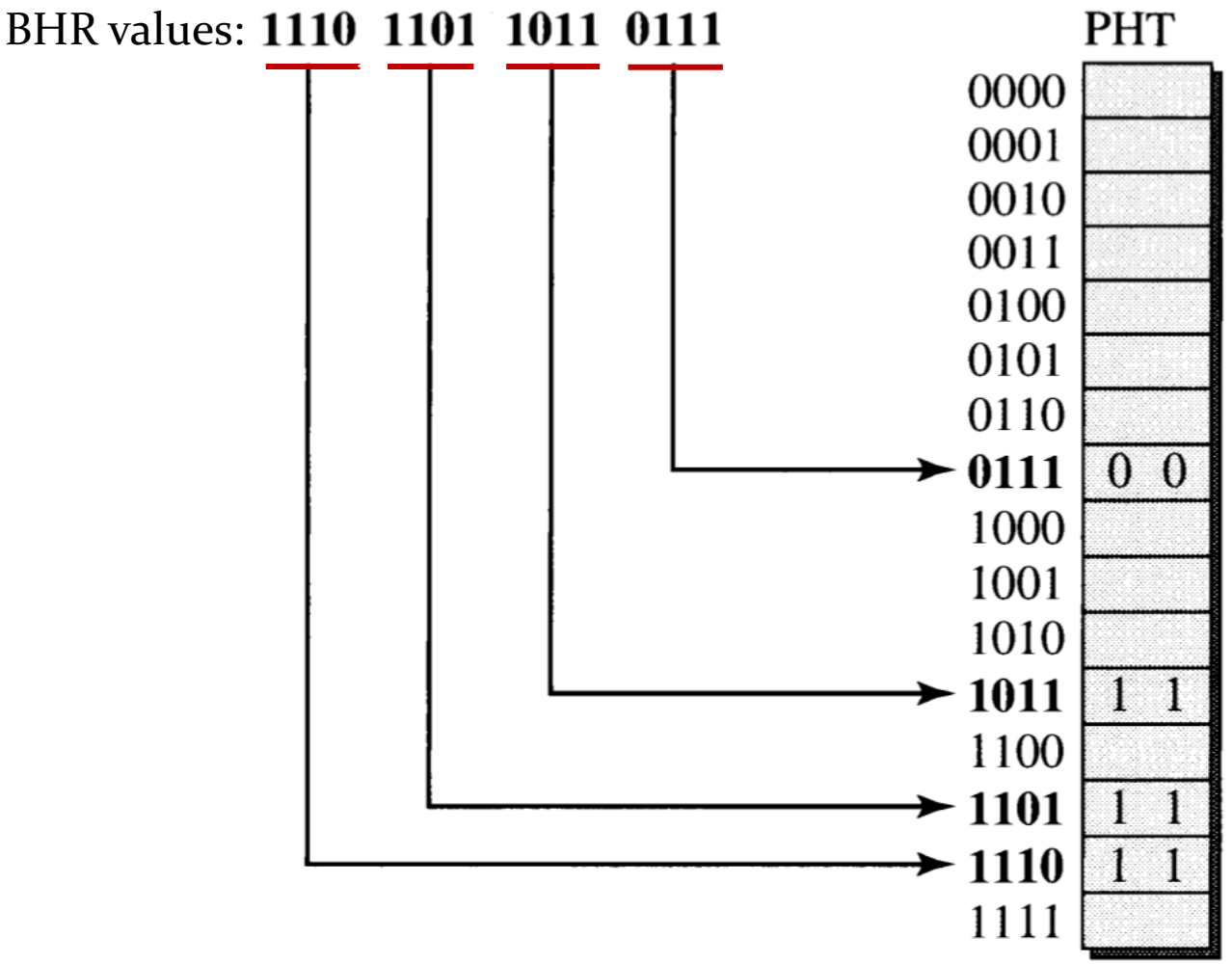
Example 2: 1110111011101

Say we only use BHR to index PHT (no PC bits)

The branch pattern repeats every four branches

If we have a PHT size 16, what happens?

We can learn this pattern exactly!



Taxonomy of branch predictors:

"Alternative Implementations of Two-Level Adaptive Branch Prediction"

Three types of Branch History Tables (BHTs)

"Global"

1) Global (G): BHT is a single BHR shared by all branches
2) Set (S): Index BHR using hash function
3) Per address (P): Index BHR using PC address

"Local"

Three types of Pattern History Tables (PHTs)

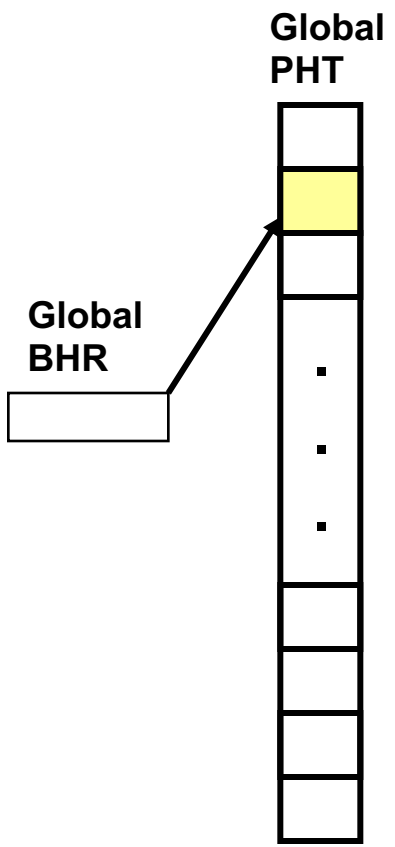
1) Global (g): 1 shared PHT, indexed by BHR
2) Set (s): PHT selected with branch hash, FSM selected w BHR
3) Per address (p): PHT selected with branch addr, FSM w BHR

"Local" and "Global"

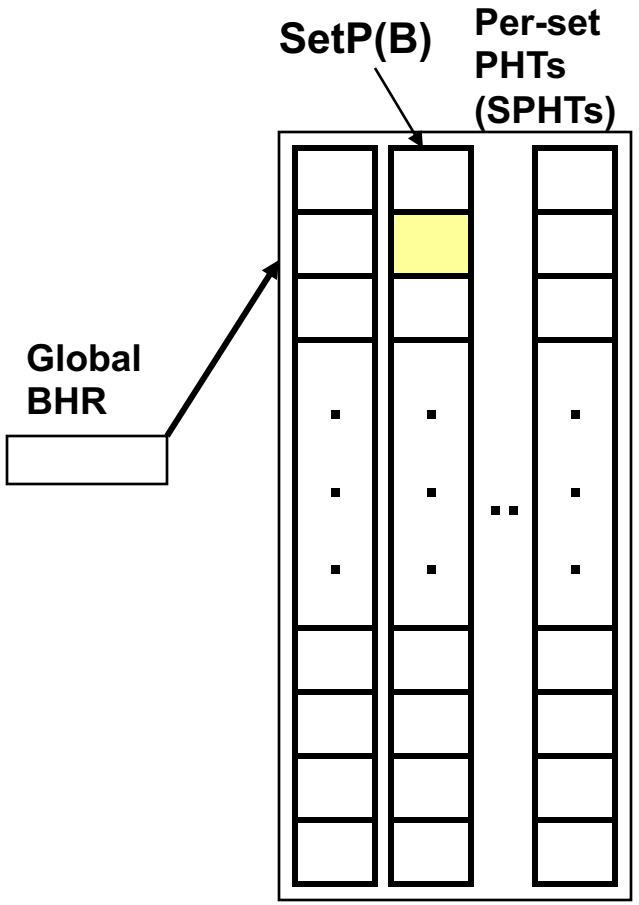
All 9 combos specified by <X>A<y> where X={G,S,P} and Y={g,s,p}

Global History Schemes

GAg

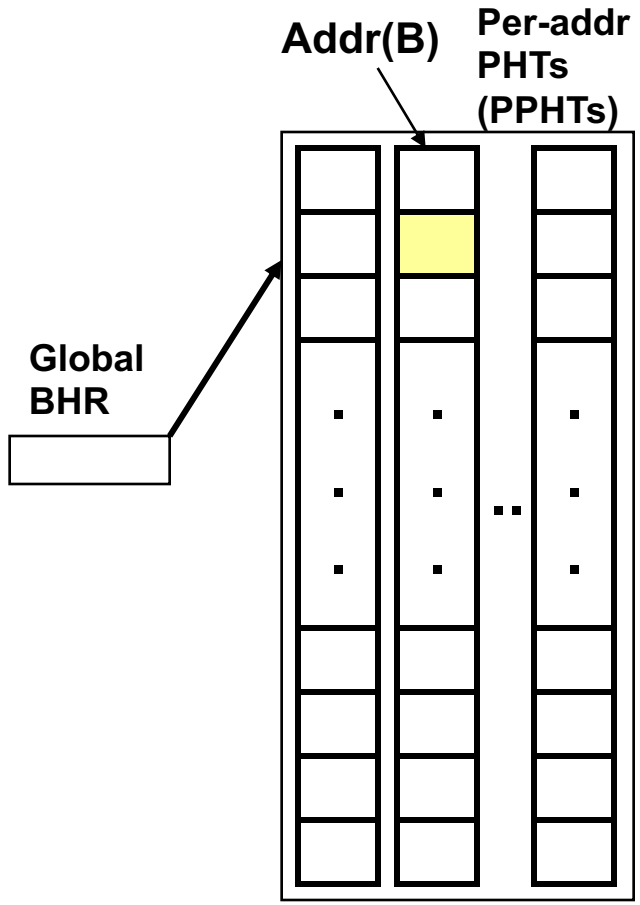


GAs

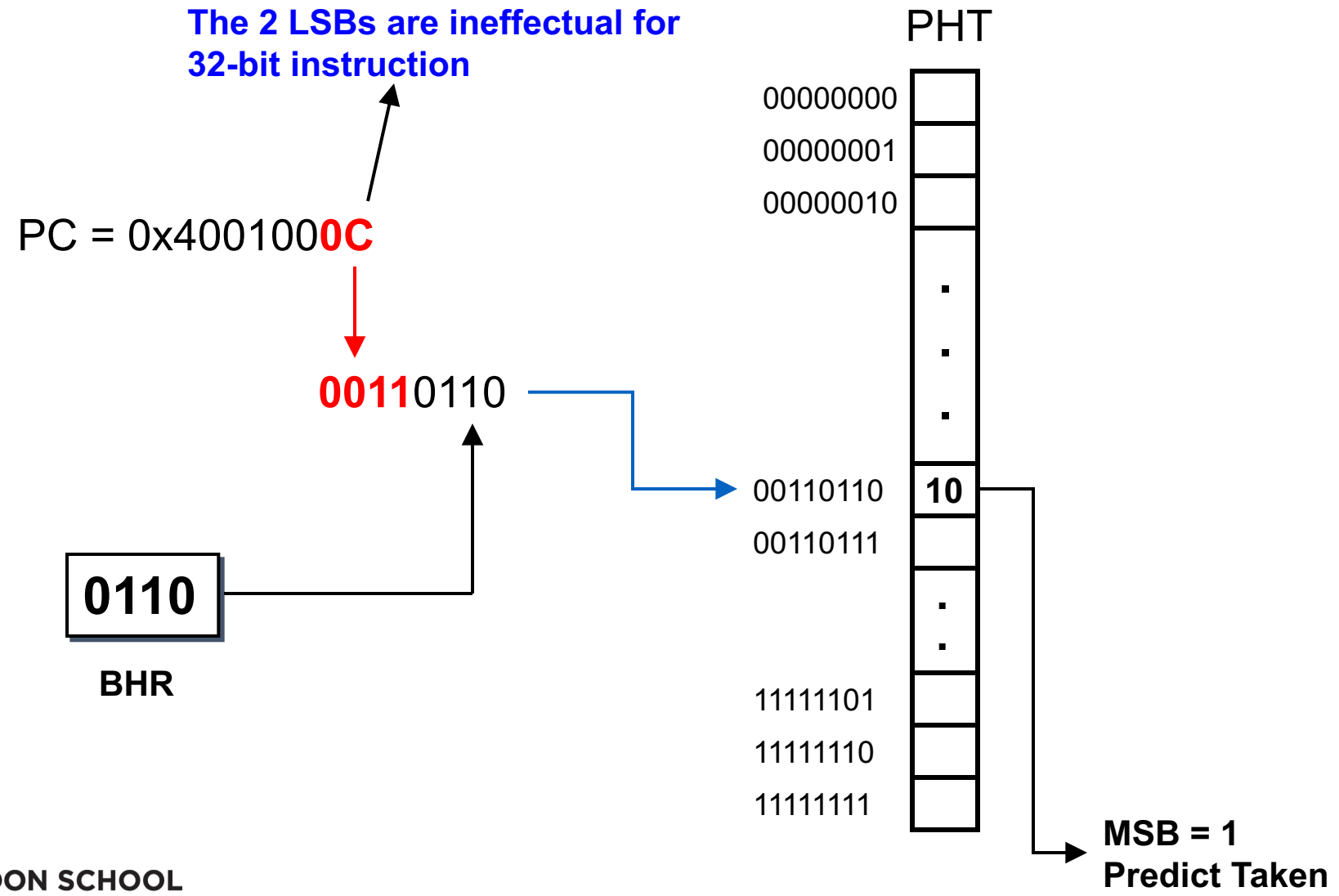


Set can be determined by branch opcode, compiler classification, or branch PC address.

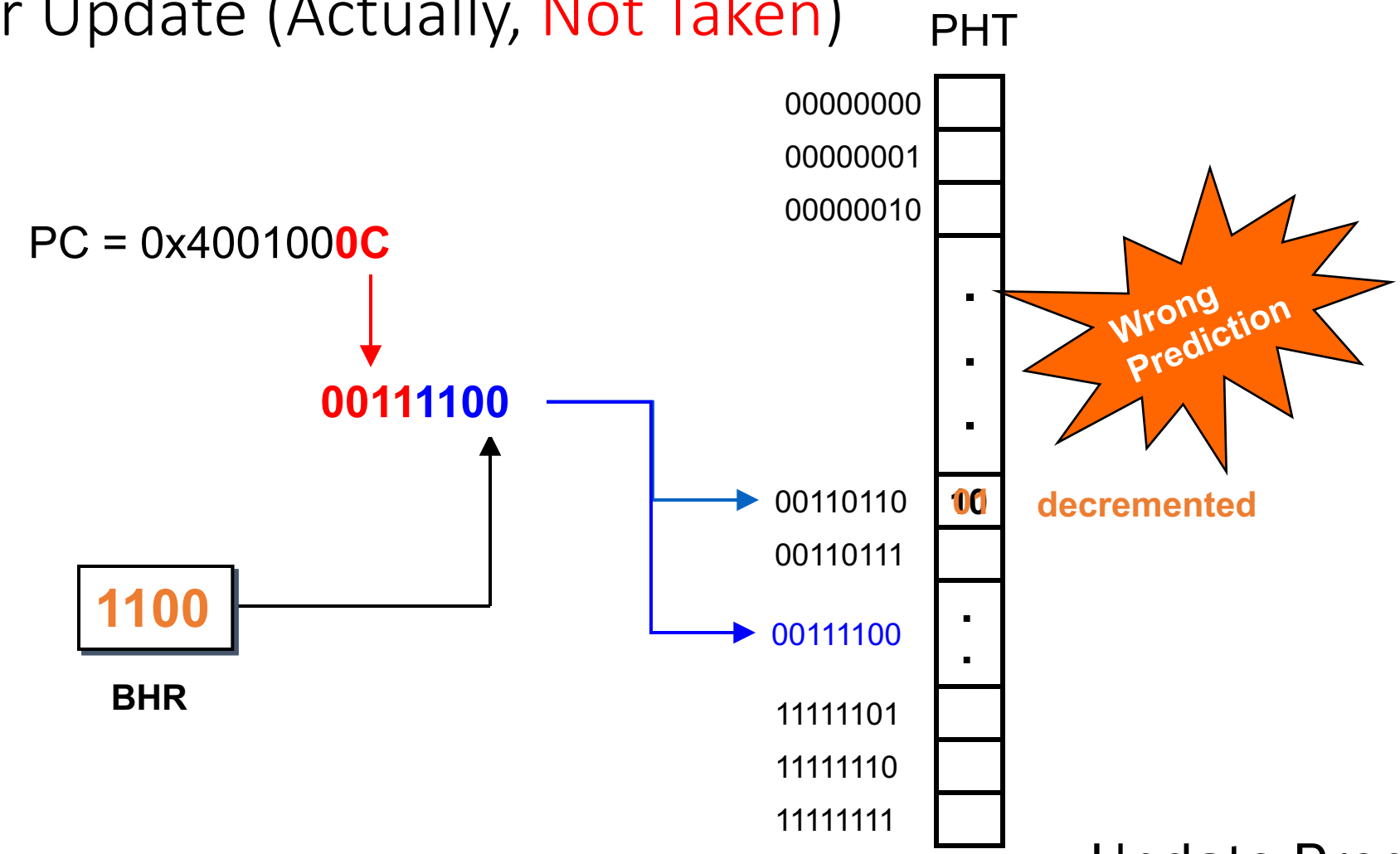
GAp



GAs Two-Level Branch Prediction

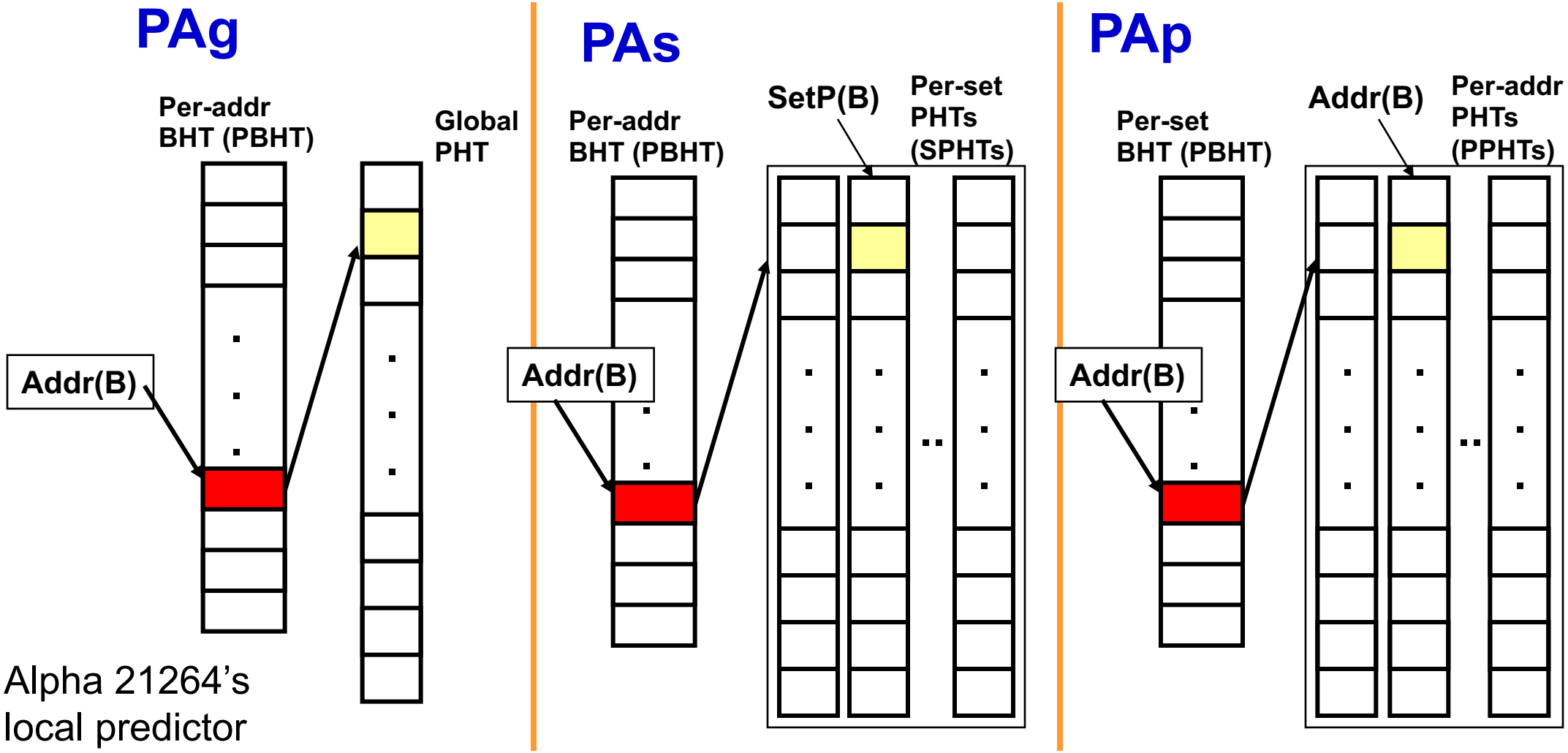


Predictor Update (Actually, Not Taken)



Update Predictor after branch is resolved

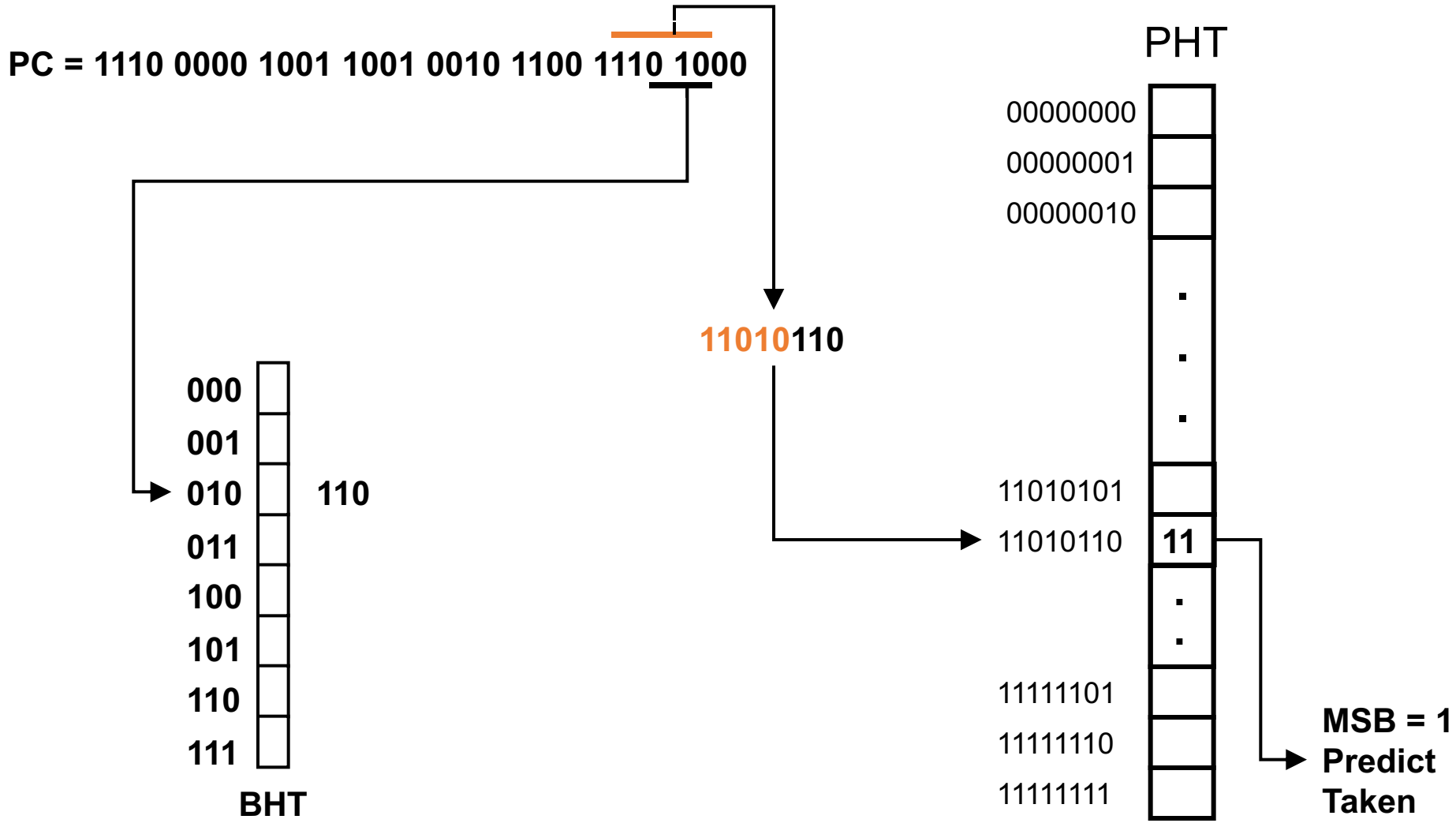
Per-Address History Schemes



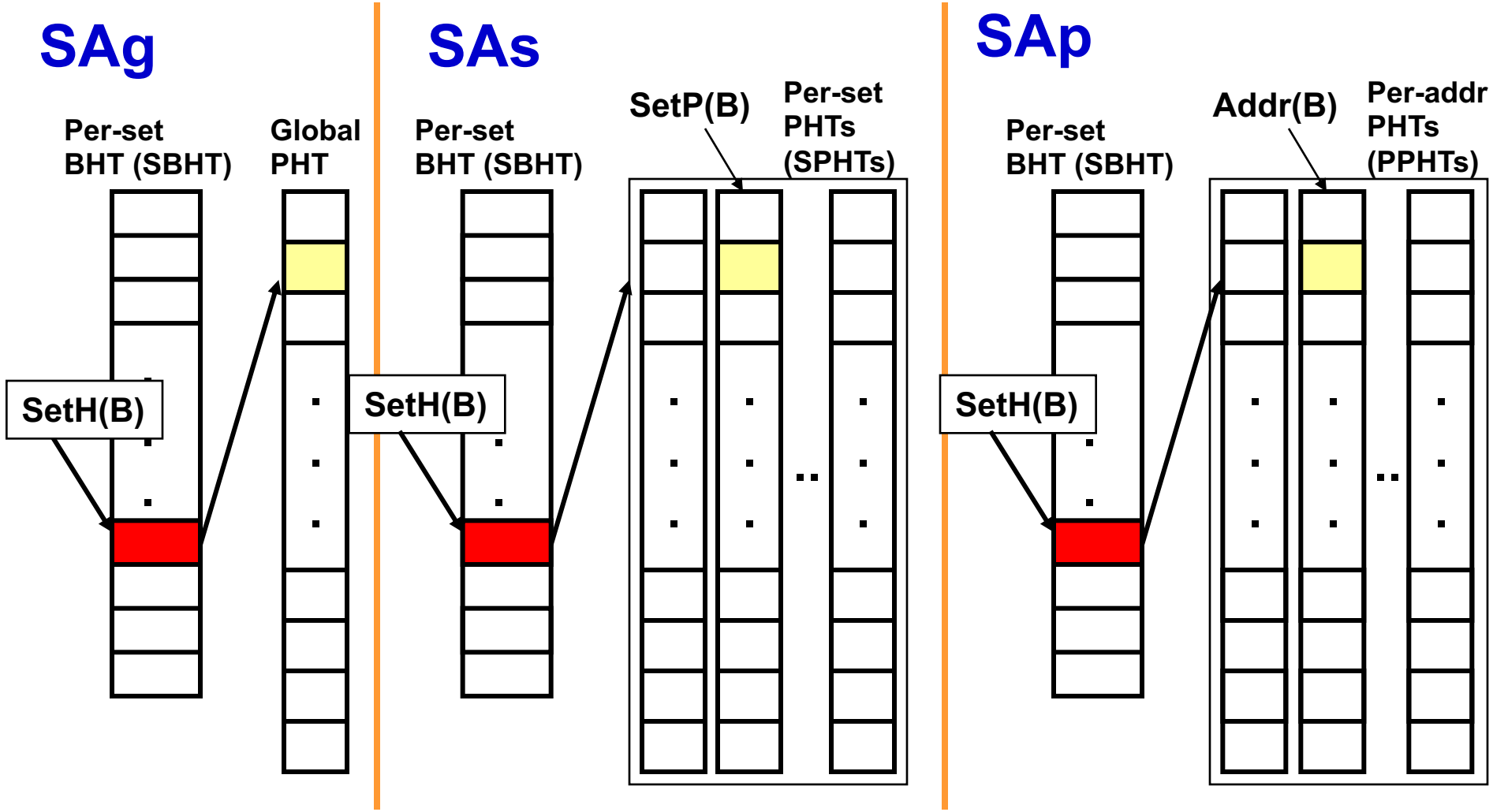
Alpha 21264's local predictor

Ex: P6, Itanium

PAp Two-Level Branch Predictor



Per-Set History Schemes



NYU TANDON SCHOOL OF ENGINEERING

More tradeoffs

- How to index PHT?
  - Using more BHR bits enables better correlation
  - Fewer PC bits causes more PHT collisions/aliasing

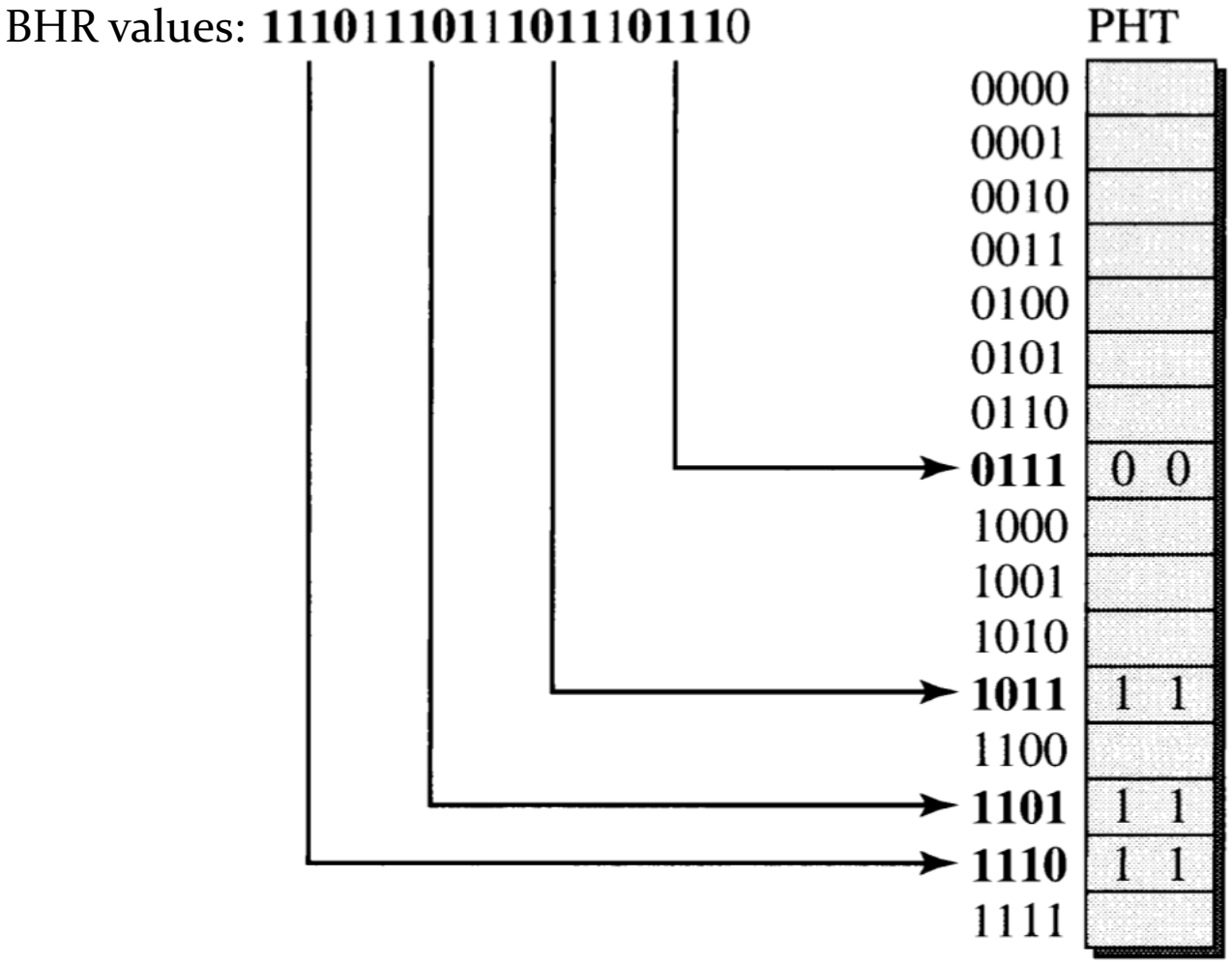
- BHR-heavy indexing performs well but:
  - Causes aliasing
  - Results in poor PHT utilization
    - Why?
    - Normally only have a few common branch patterns, rest unused

Recall example

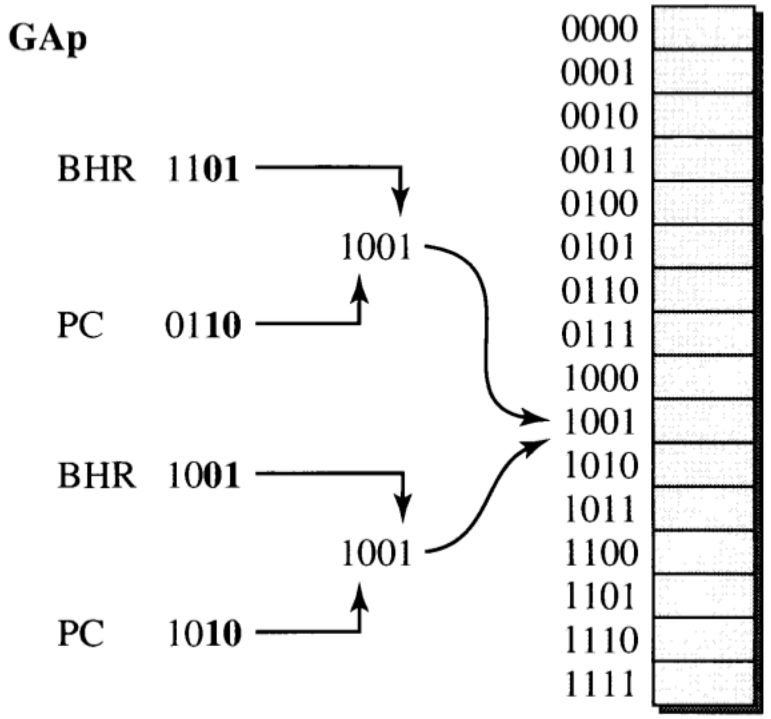
Only using 4 of 16 slots..
Poor utilization!
We can do better!

Q: What happens if another branch aliases to the same PHT entries?

Everything works!
"Neutral Interference"



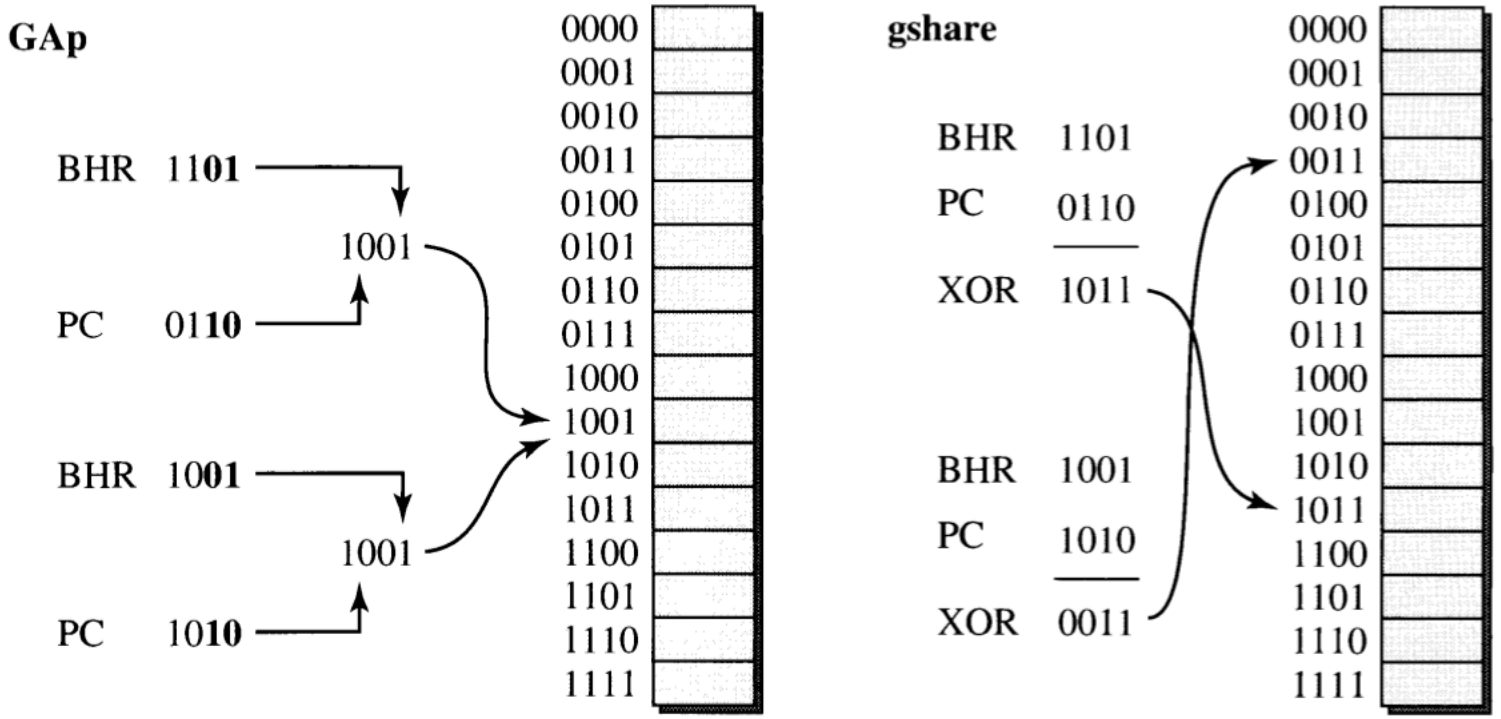
Index-sharing: the gshare predictor



Index-sharing: the gshare predictor

Pros
+ Better utilize PHT
+ Resolves conflicts

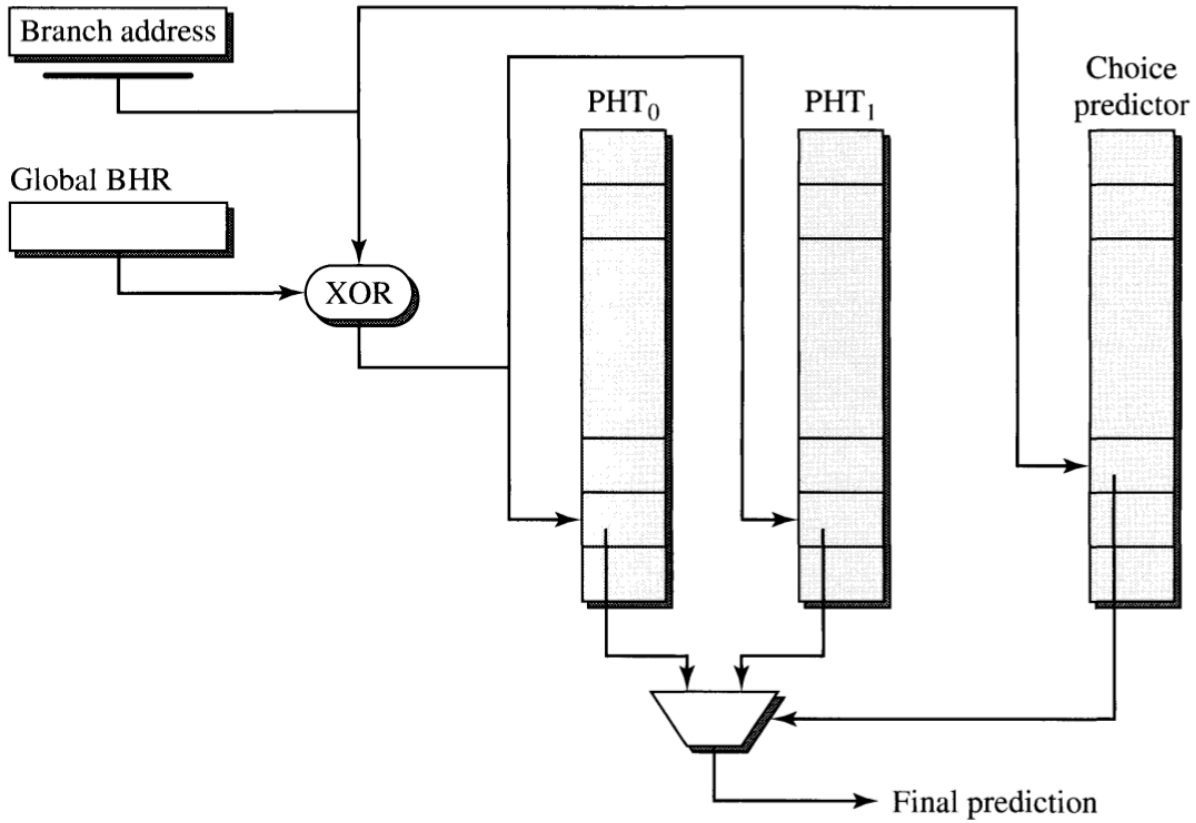
Cons
- Can cause new conflicts
- Minor increase to critical path



Bi-Mode predictor

- Most branches are biased toward T/NT
  - PHTs learn bias
  - Choice predictor learns branch bias
- Turns negative conflicts into neutral
  - If collide in the same PHT,
    Choice made it happen, so probably good

- Updating
  - Selected PHT always updated
  - Not selected bank not updated
  - Choice predictor
    - Always updated using branch direction
    - Unless Choice Pred. outcome was wrong but selected PHT's guess was correct



The gskew predictor

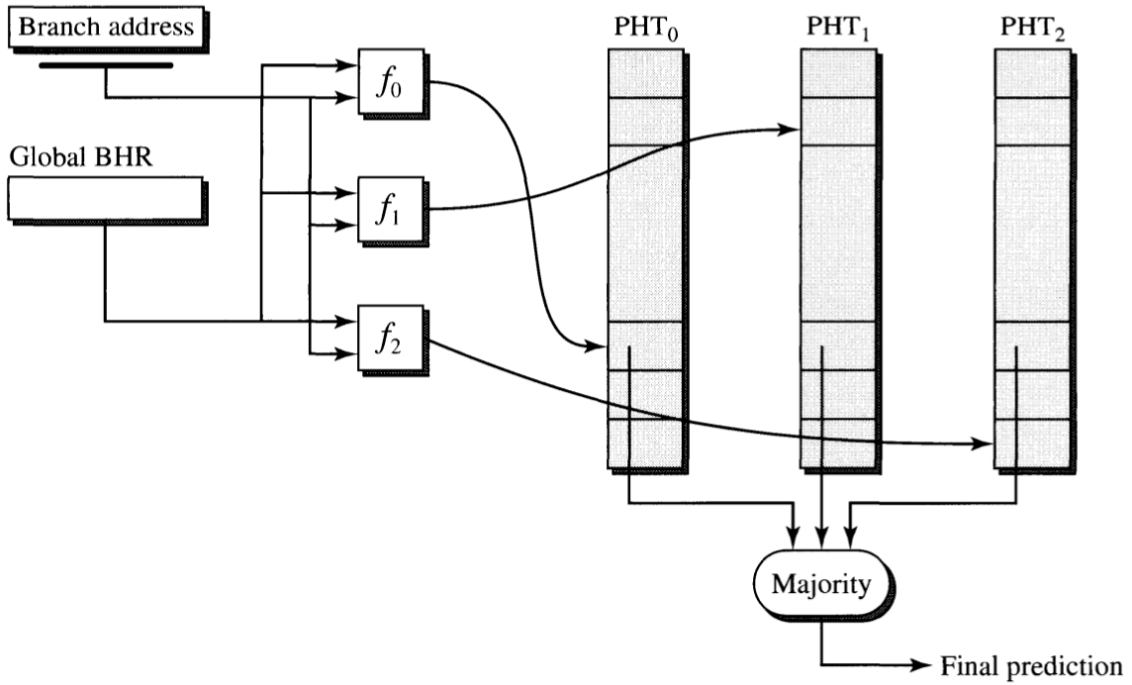
- Create 3+ PHT "banks"
- Hash Address-BHR pair
  - How does this compare to Bi-Mode?
- Read predictions from each banks, take majority
- Intuition:
  - Even if you have a conflict, the majority vote will mitigate negative impact
- Clever trick: fo, f1, f2 can be created such that:
  if fo(x) == f1(x), then f1(x) != f2(x), f2(x) != fo(x)

  What does this mean? If you have multiple PHT conflicts they're from independent branches

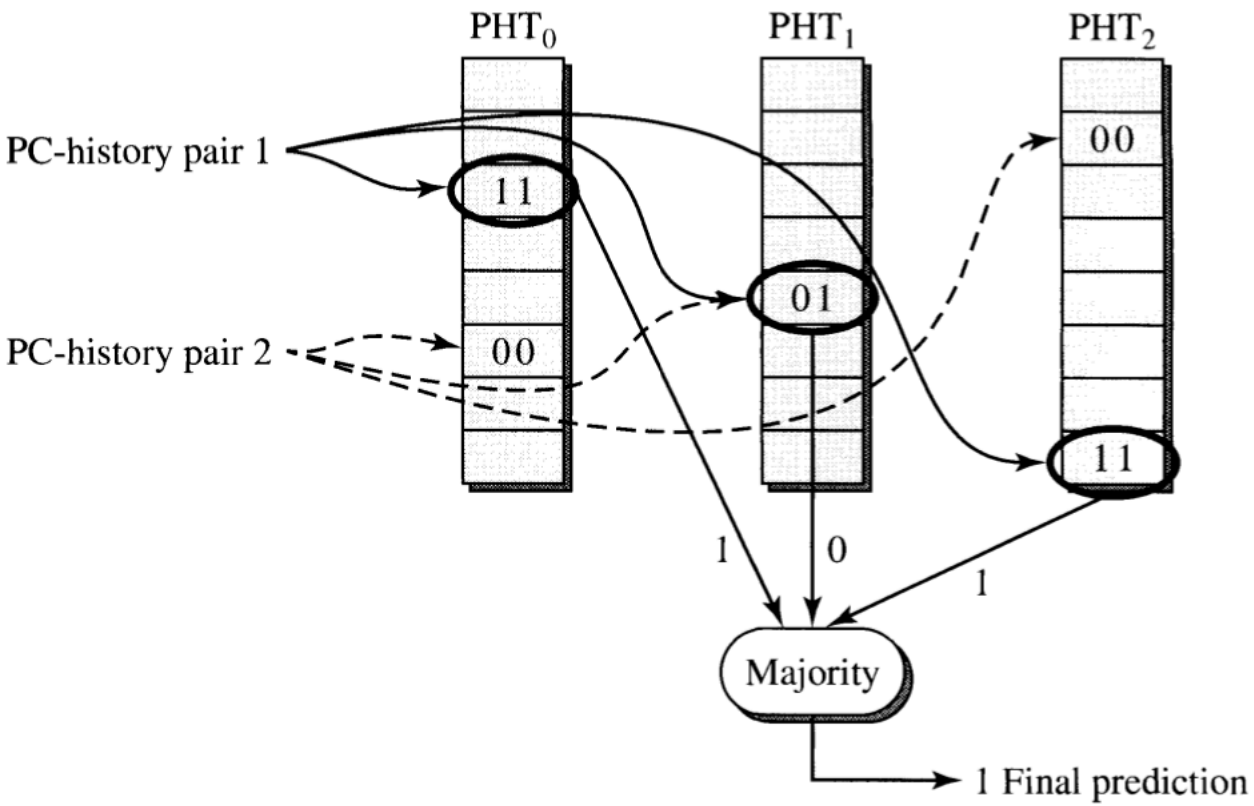
Two update policies:

1) Total update: update all PHTs after branch resolves
2) Partial update: If prediction correct but PHT wrong, don't update.

Partial tends to have best performance.

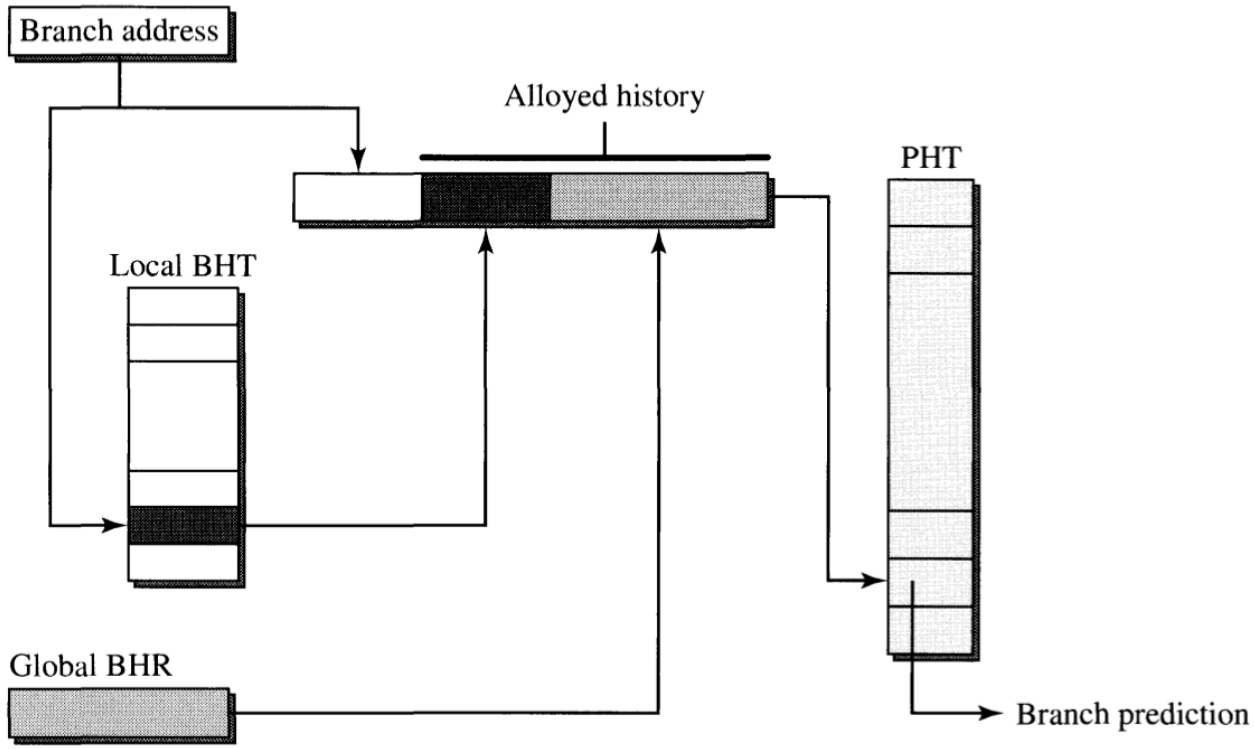


Gskew example



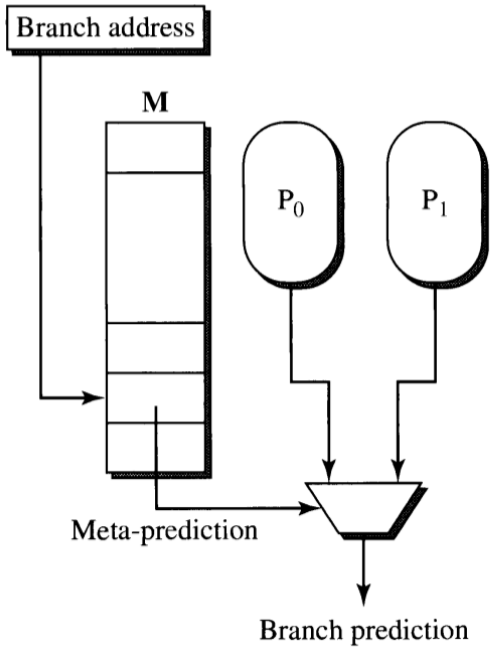
The Alloyed History Predictor

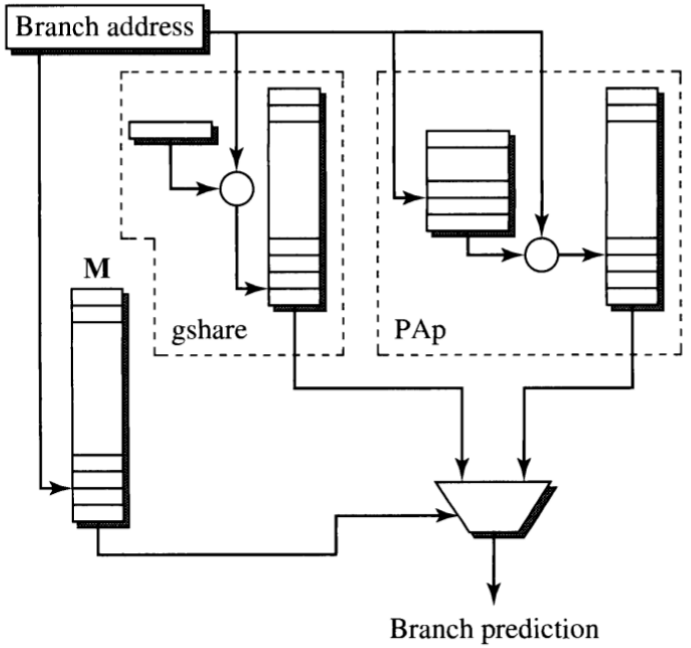
- Some branches need global history (1 BHR) some need local history (BHT)
  What to do?
  - Combine them!
- Alloyed predictor tracks both global and local branch history
- Three benefits:
  - Cheap!
  - Best of both scopes
  - Can correlate even more, since some branches need context from both



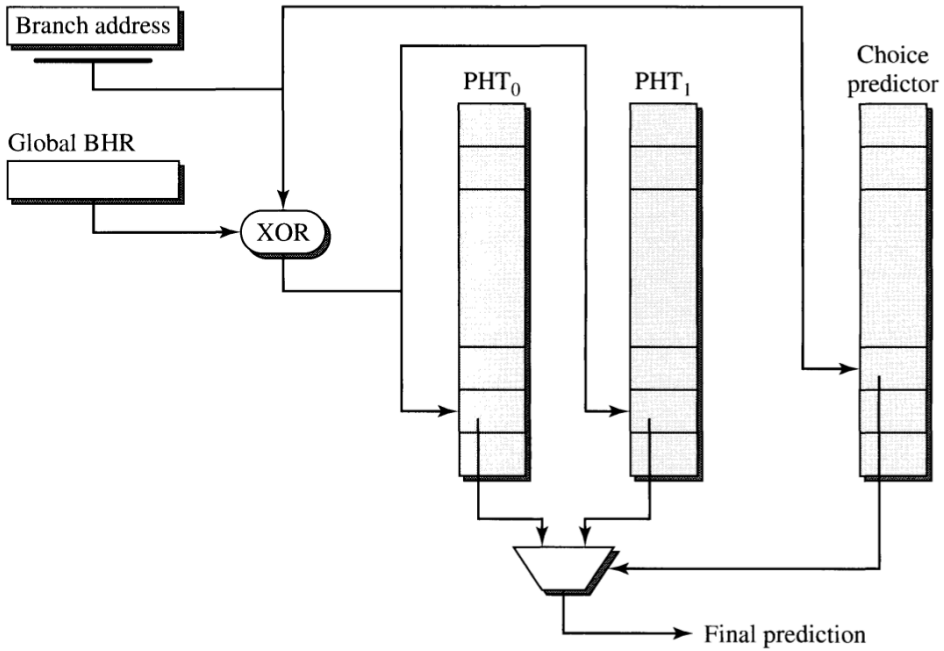
Tournament predictors

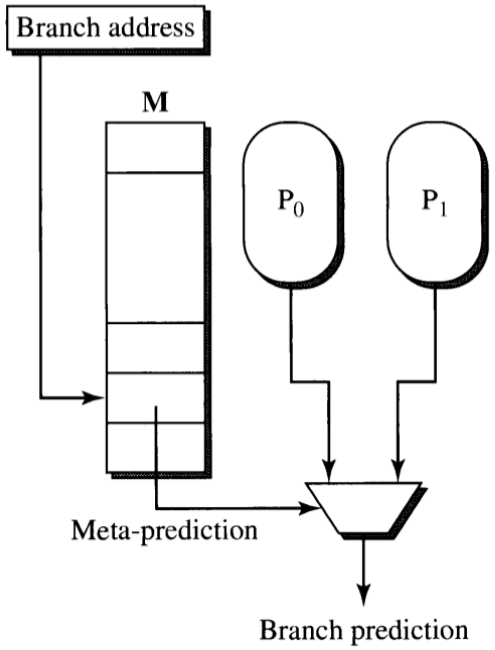
- Combine multiple BPs
- Use any 2 XAy BPs
- M is 2b counters for MUX select
- After branch resolved, update both P0, P1
- Update M based on predictor mismatch, favoring correct
  - Increment if P0 right, P1 wrong
  - Decrement if P0 wrong, P1 right
  - If both right/wrong do nothing
- How is this different from Bi-Mode?

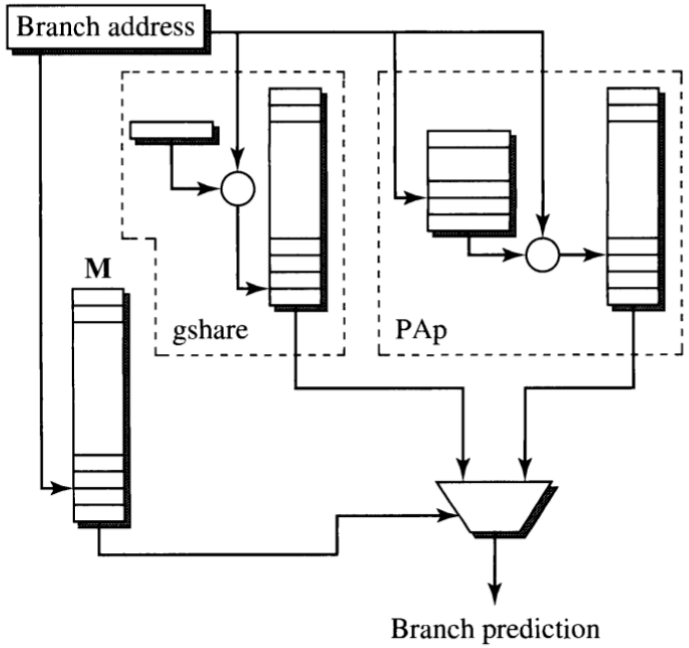




Bi-mode and Tournament







What's happening today

Current state-of-the-art (rumored to be what Intel is using)

"TAGE: TAgged GEometric predictor", (Seznac, JILP 2006)

What's research looking into?

Neural predictors!

1) Daniel Jimenez (Texas A&M) did lots of pioneering work
2)Yale Patt(?) has neural prediction paper at MICRO this month:

"BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches"

This sounds interesting.. I want to learn more

1) Highly suggest reading

   Chapter 9 (Advanced Instruction Flow Techniques) in "Modern Processor Design" by Mikko Lipasti

2) Read "TAGE" and work backwards through references

3)Try out the branch prediction championship:

   https://www.jilp.org/cbp2016/

4) If you can beat the SOTA in branch prediction, I'll give you a PhD.

“Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions” (IISWC 2019)

55% performance hit from miss prediction at 4x scaling

This performance benefit on par with process technology node

New foundries (chip making plants) cost tens of billions of dollars!

~$20B by some estimates

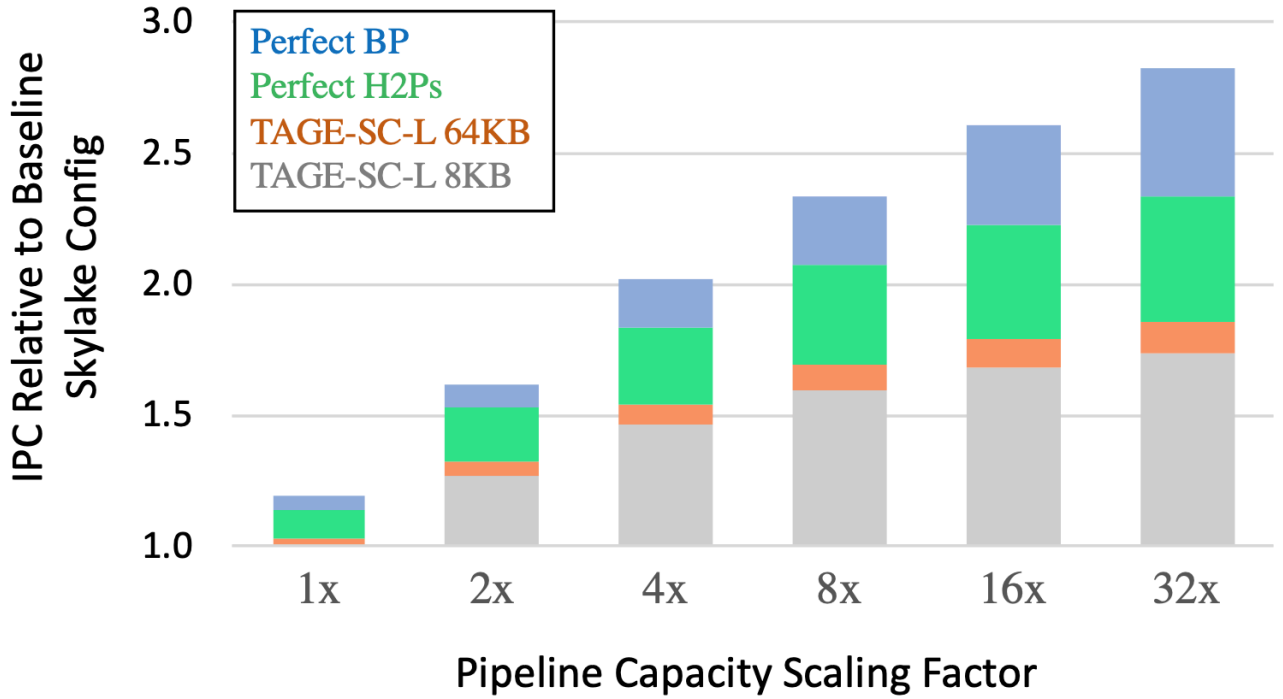
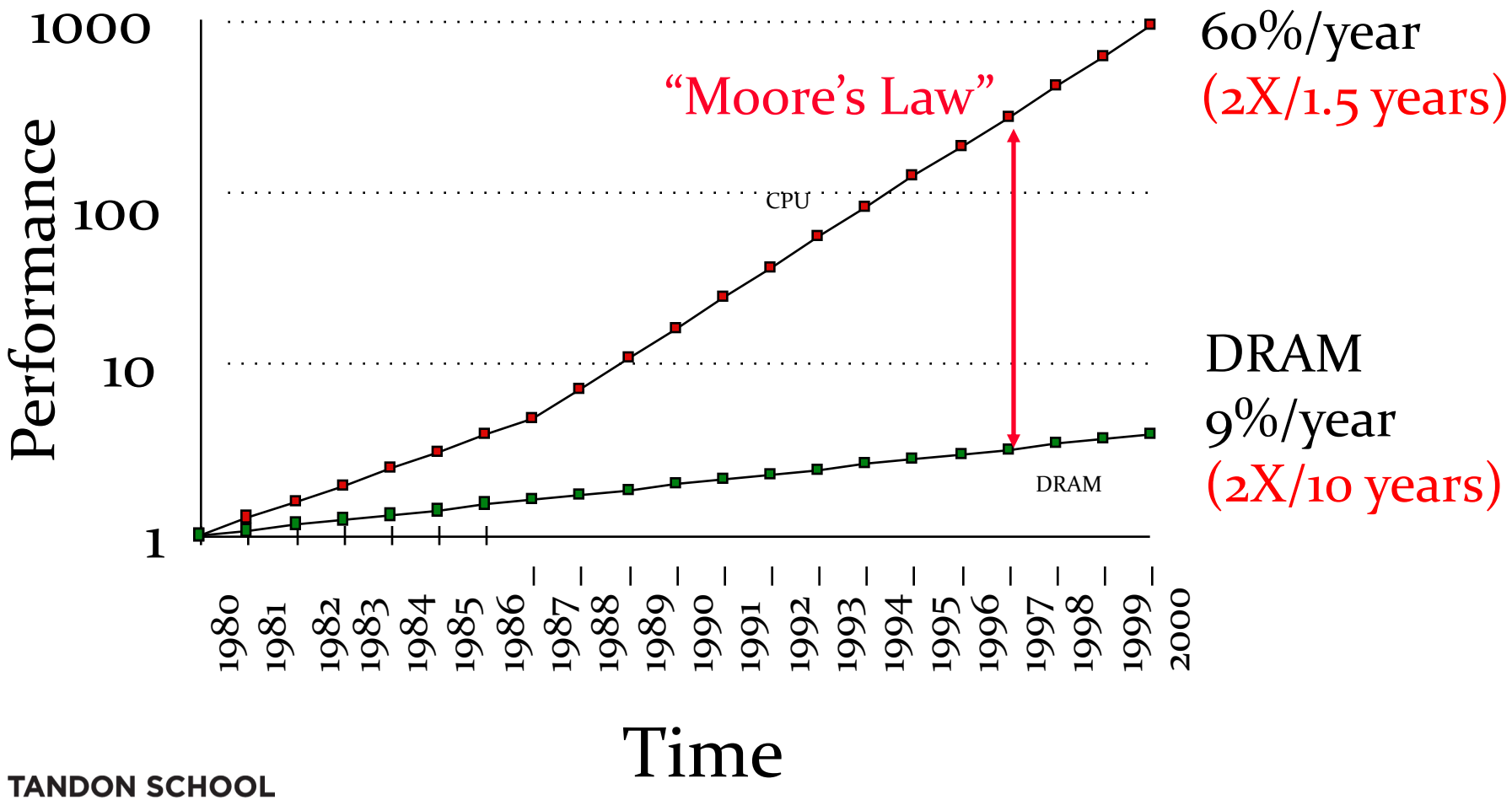


Fig. 1: Without better branch prediction, scaling the pipeline capacity of an Intel Skylake configuration will produce diminishing returns in single-threaded IPC for SPECint 2017.

Caches

Why Care About Memory Hierarchy?

Processor-DRAM Performance Gap grows 50% / year



Memory Issues

Latency

- Time to move through the longest circuit path
- Time to read a word (from the start of request to the response)

Bandwidth

- Number of bits transported at once
- E.g., Gbs/second

Capacity

- Size of memory (32b vs. 64b max verses how much you actually have)

Power/Energy

- Cost of accessing memory (to read and write)
- Computers spend most their energy moving things around

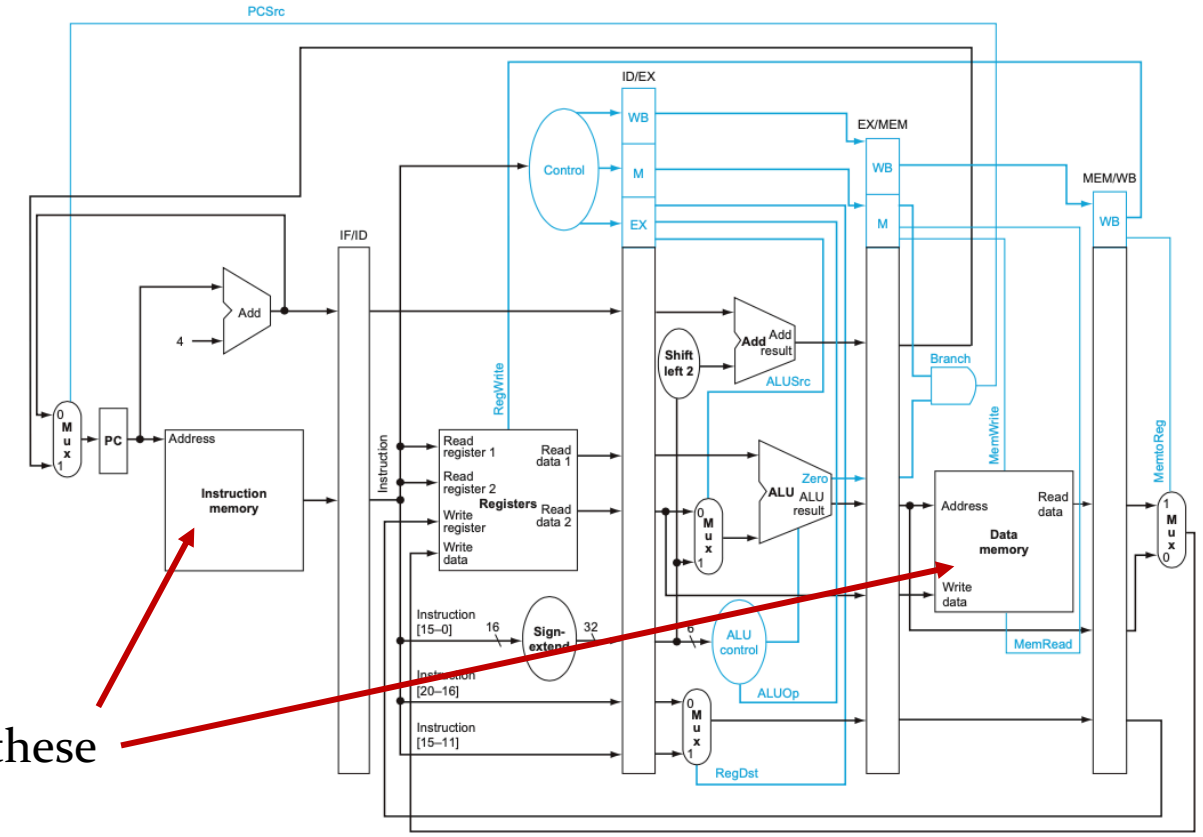
What can we do about it?

Tennent of architecture: “Smaller is faster.”

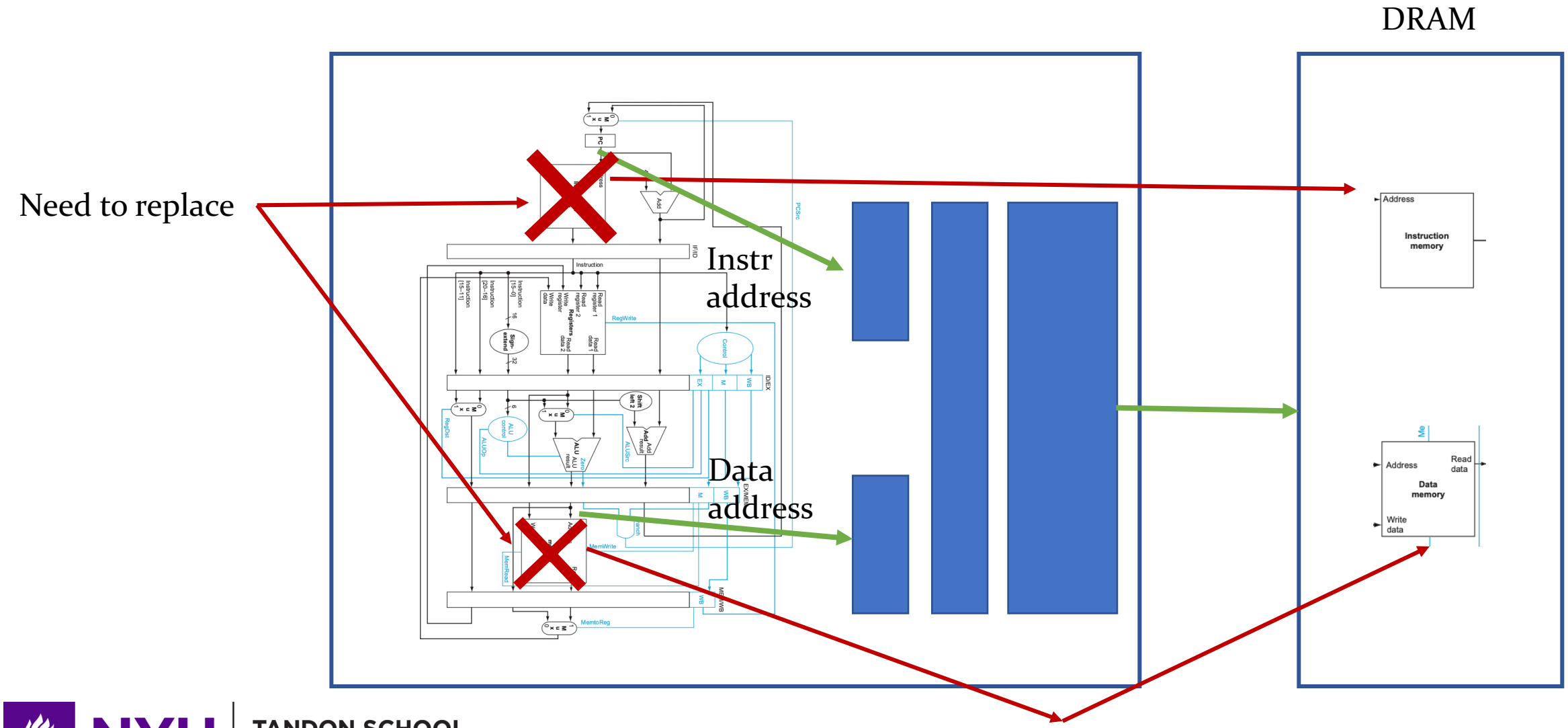
How would you solve this?
“Temporary” buffers between the register file and memory.

microArchitecture or Architecture technique?

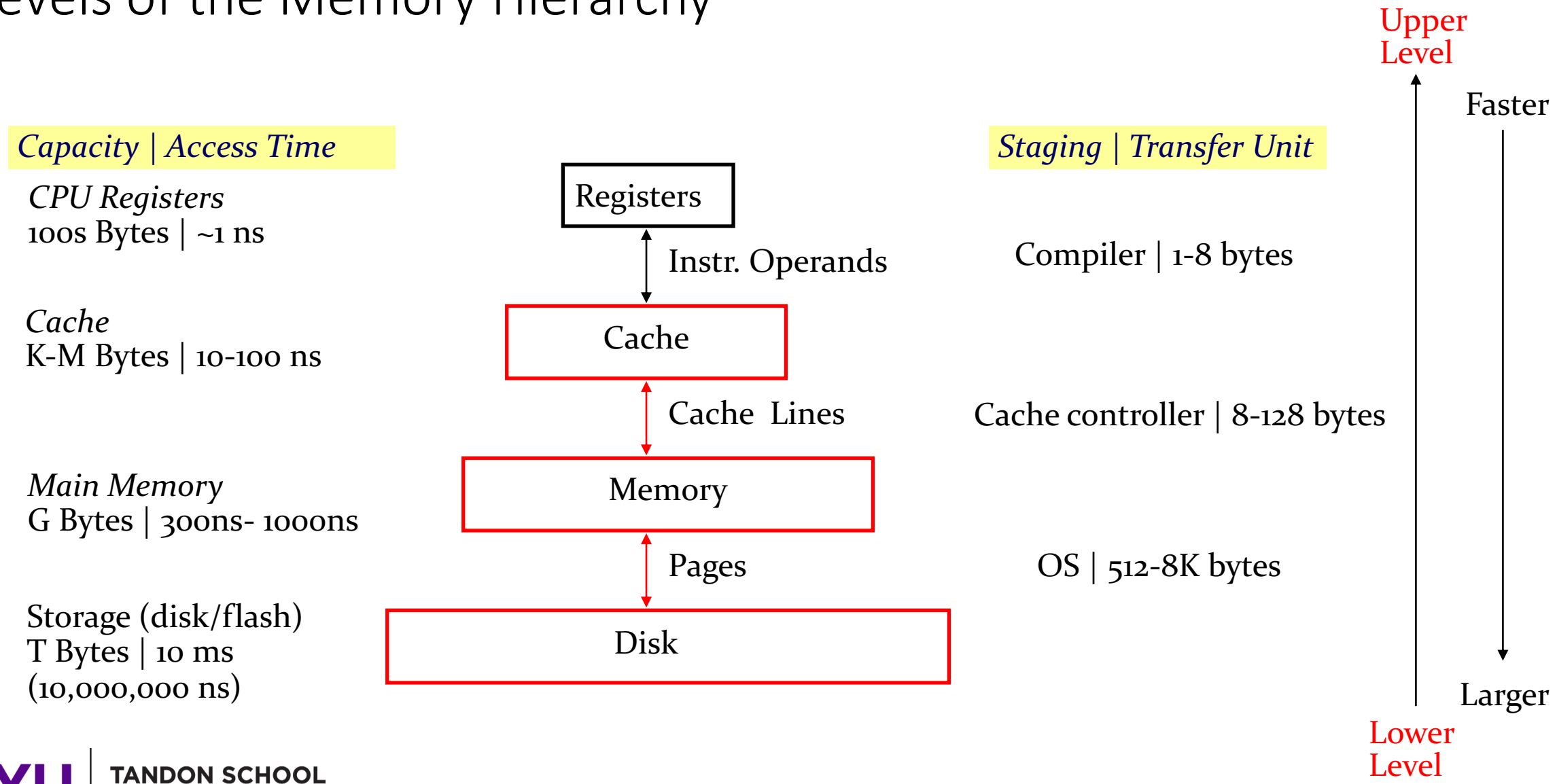
What can we do about it..



What can we do about it..



Levels of the Memory Hierarchy



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?

Why do Caches Work: Principle of Locality

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., straightline code, array access

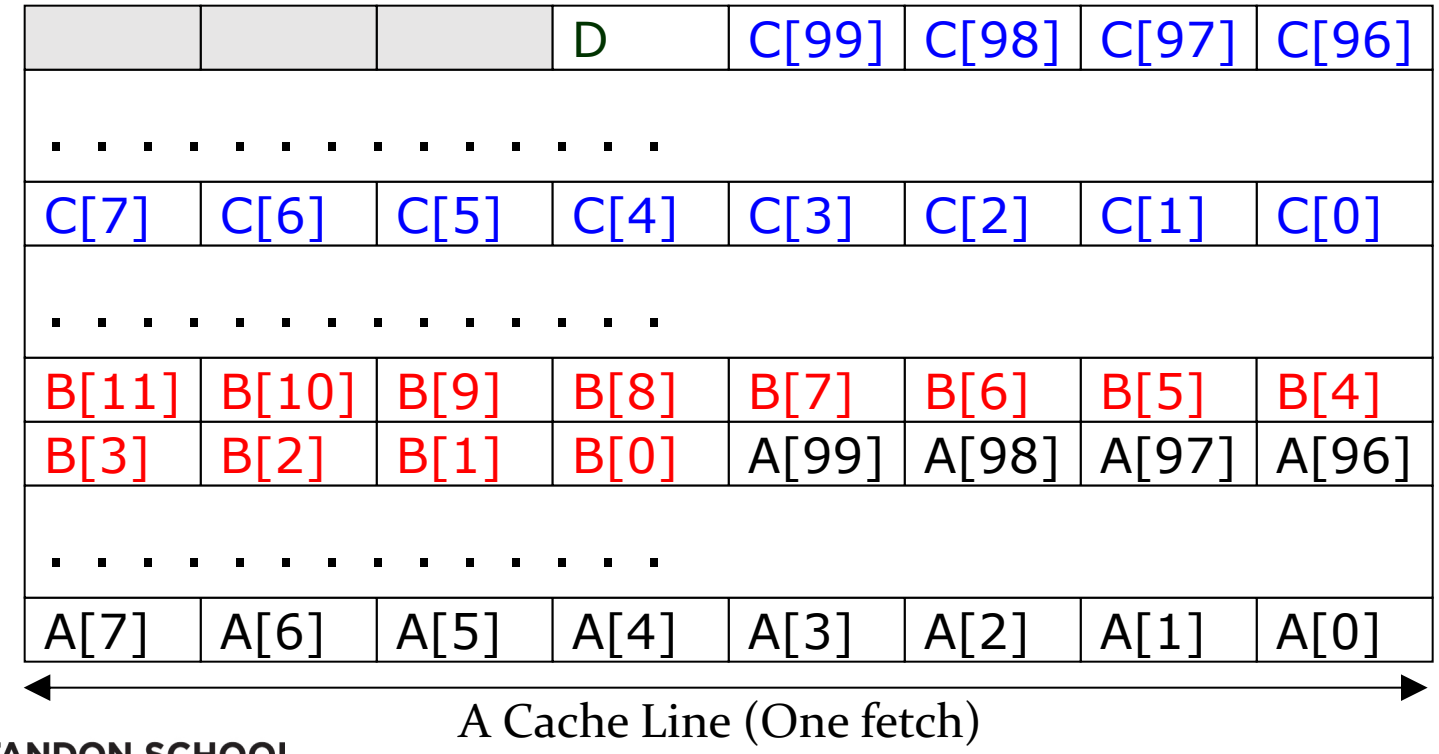
Traditionally, HW has relied on locality for speed

Locality is a program property that is exploited in machine design.

Example of Locality

```
int A[100], B[100], C[100], D;
for (i=0; i<100; i++) {
        C[i] = A[i] * B[i] + D;
}
```

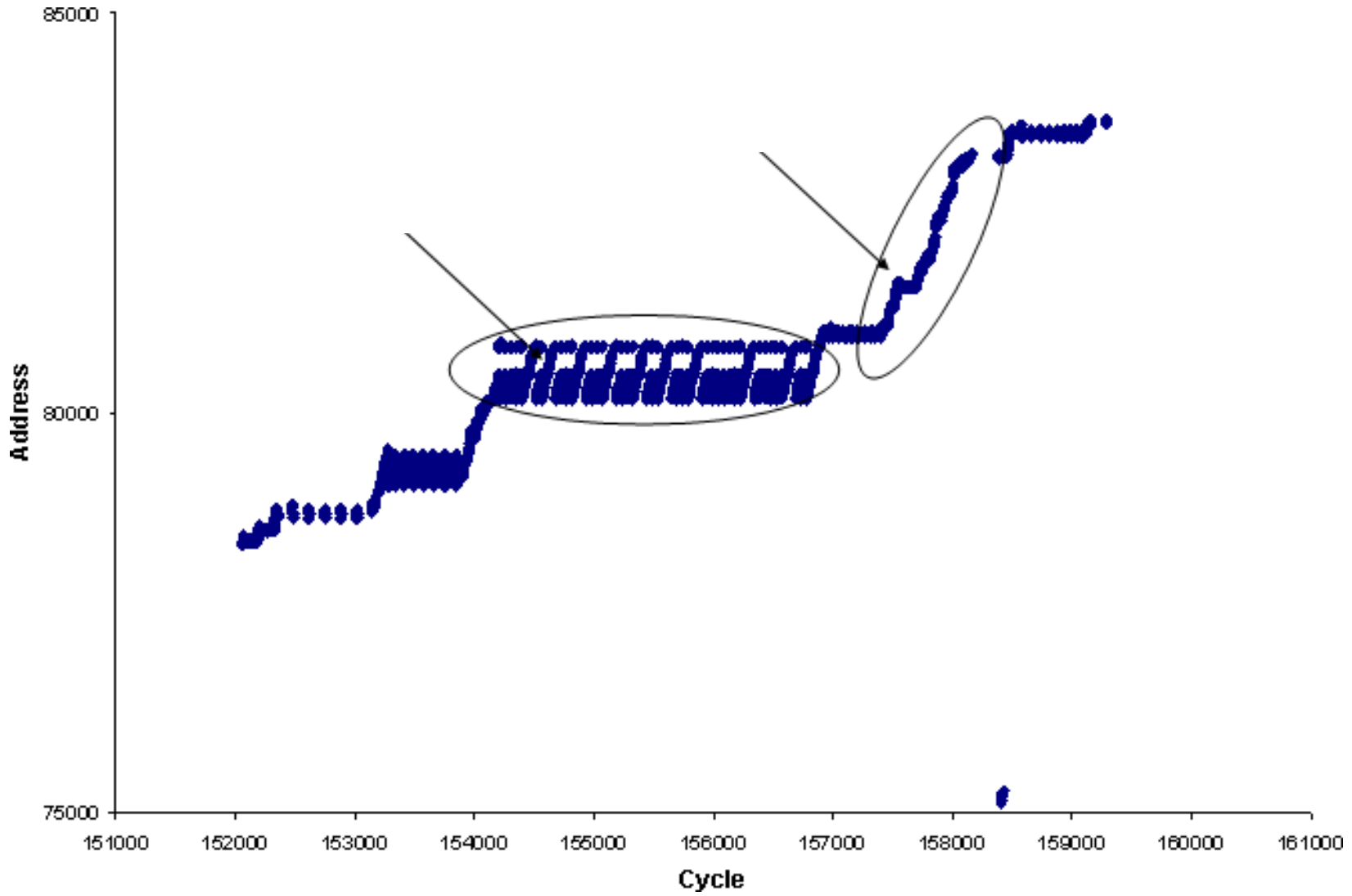
2D view of memory



What's temporal?

Where's spatial?

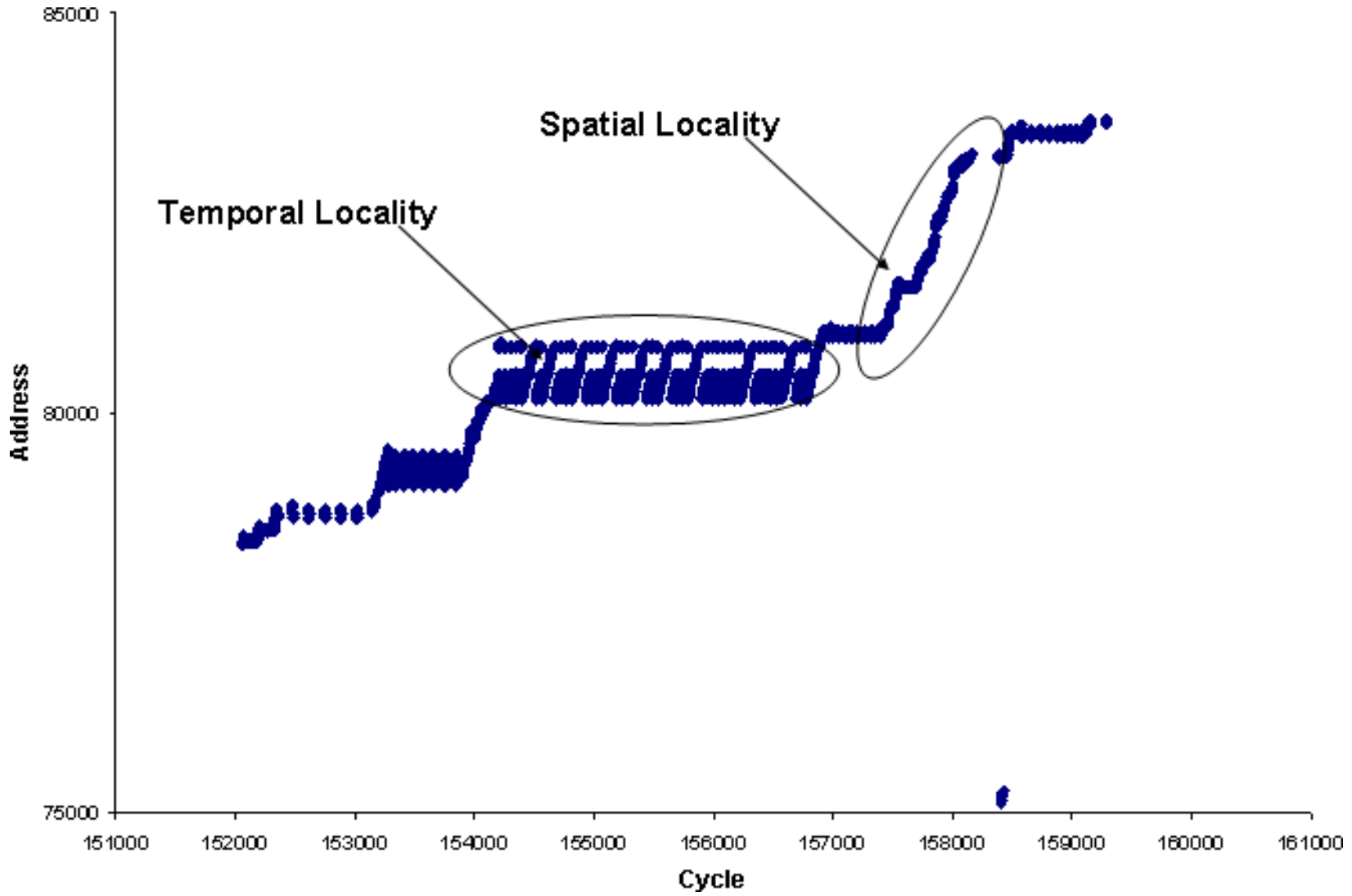
Access Pattern Exhibiting Locality



What's happening in these regions?

http://www.eetimes.com/document.asp?doc_id=1275470

Access Pattern Exhibiting Locality



What's happening in these regions?

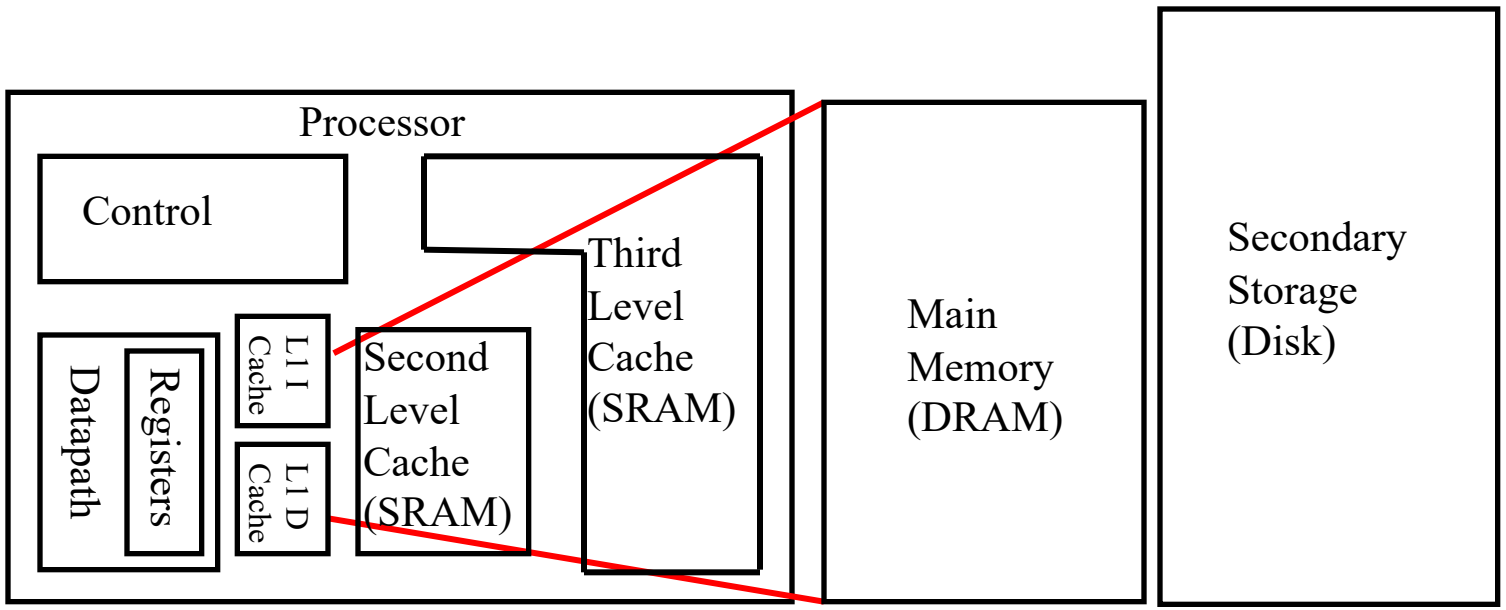
http://www.eetimes.com/document.asp?doc_id=1275470

“Memory Hierarchy” or “memory subsystem”

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

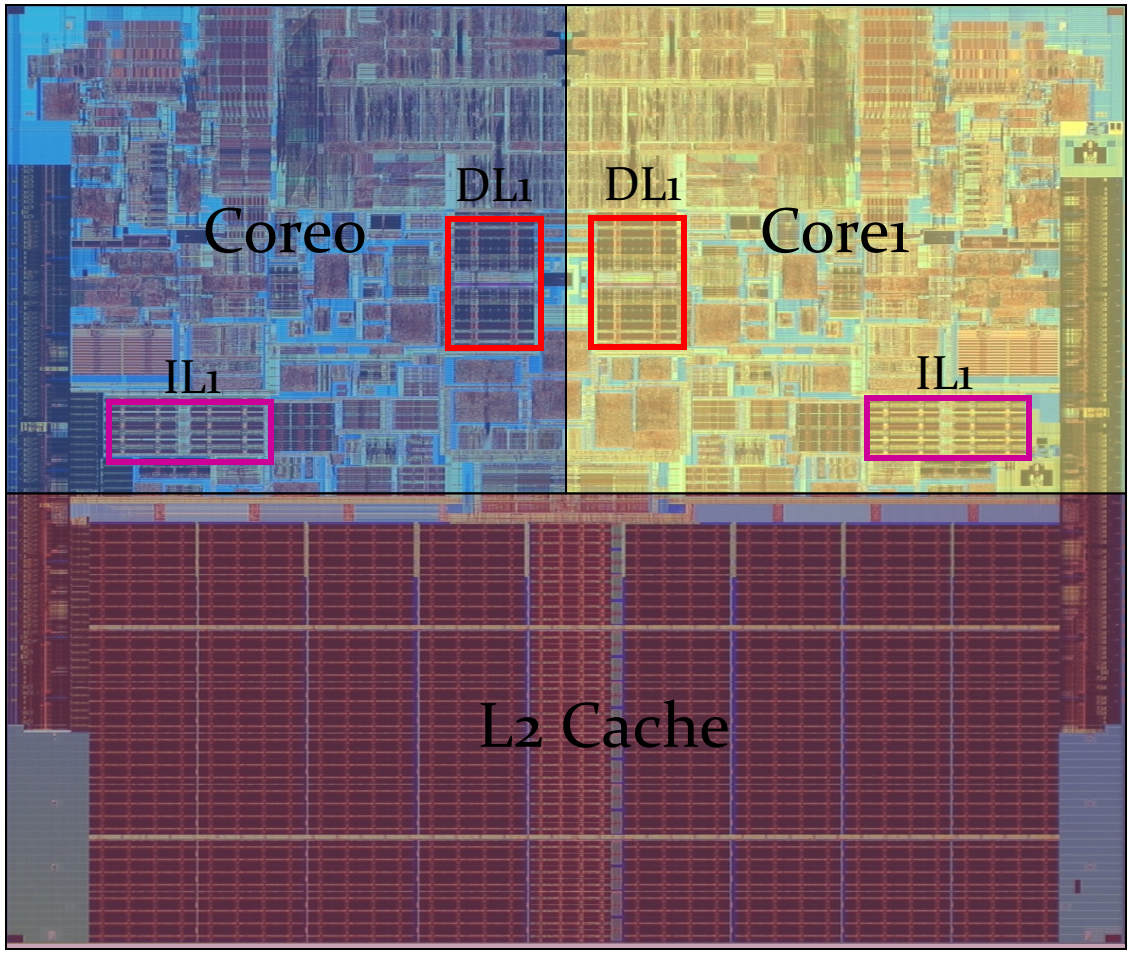
Taking advantage of locality gives the best of both worlds:

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



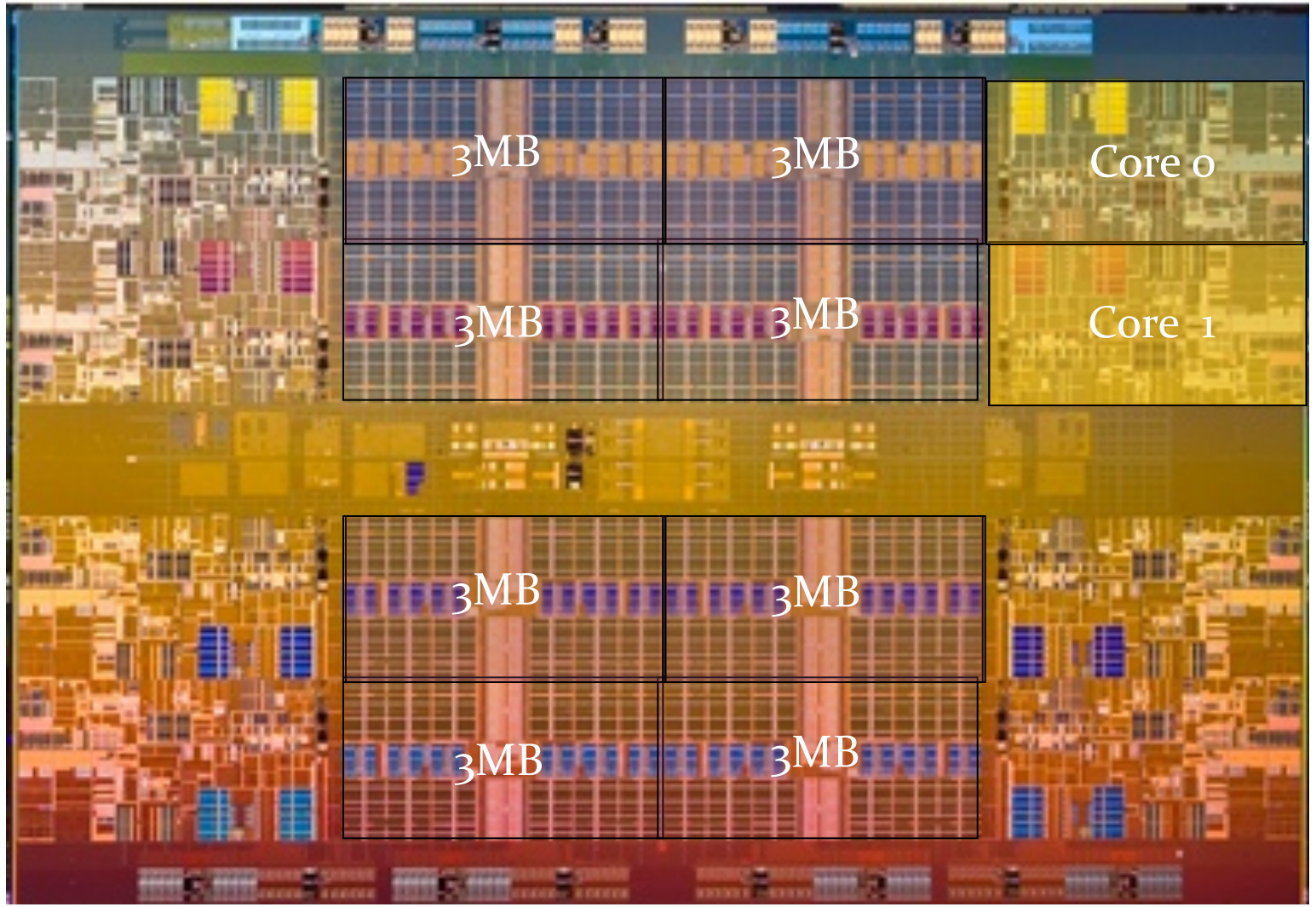
Example: Intel Core2 Duo

L1	32 KB, 8-Way, 64 Byte/Line, LRU, WB 3 Cycle Latency
L2	4.0 MB, 16-Way, 64 Byte/Line, LRU, WB 14 Cycle Latency



Source: http://www.sandpile.org

Intel Nehalem



24MB L3!

What's the trend?

More area to cache!

Cache Terminology

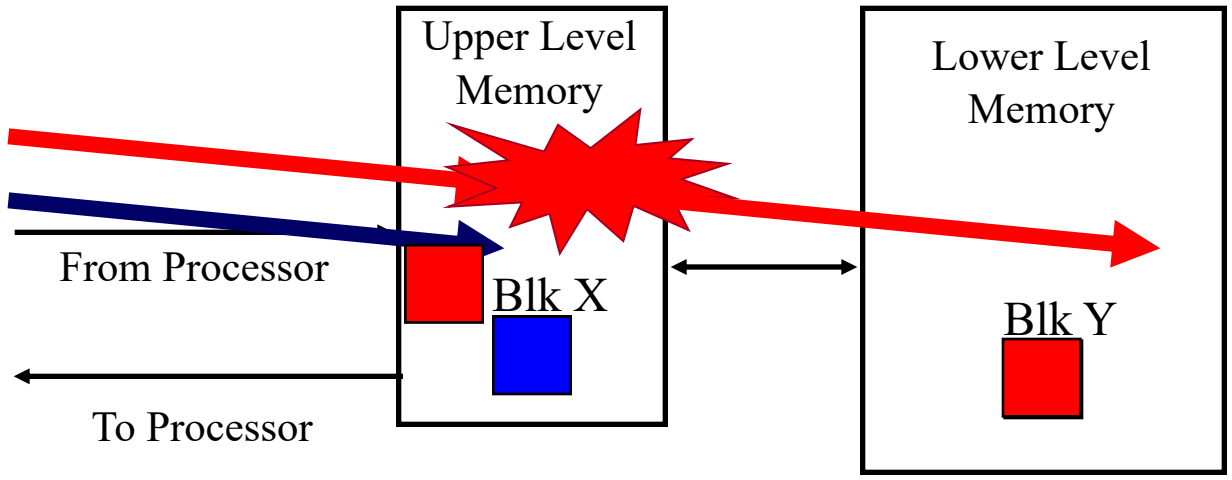
Hit: data is resident in the cache

- 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



Average Memory Access Time (AMAT)

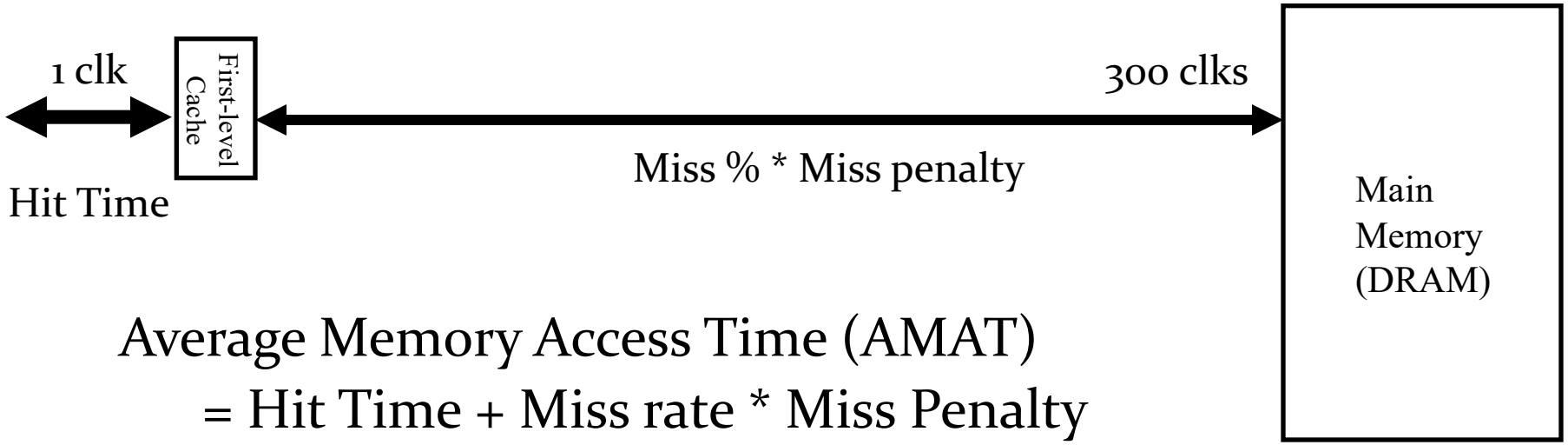
Average memory-access time

- 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)

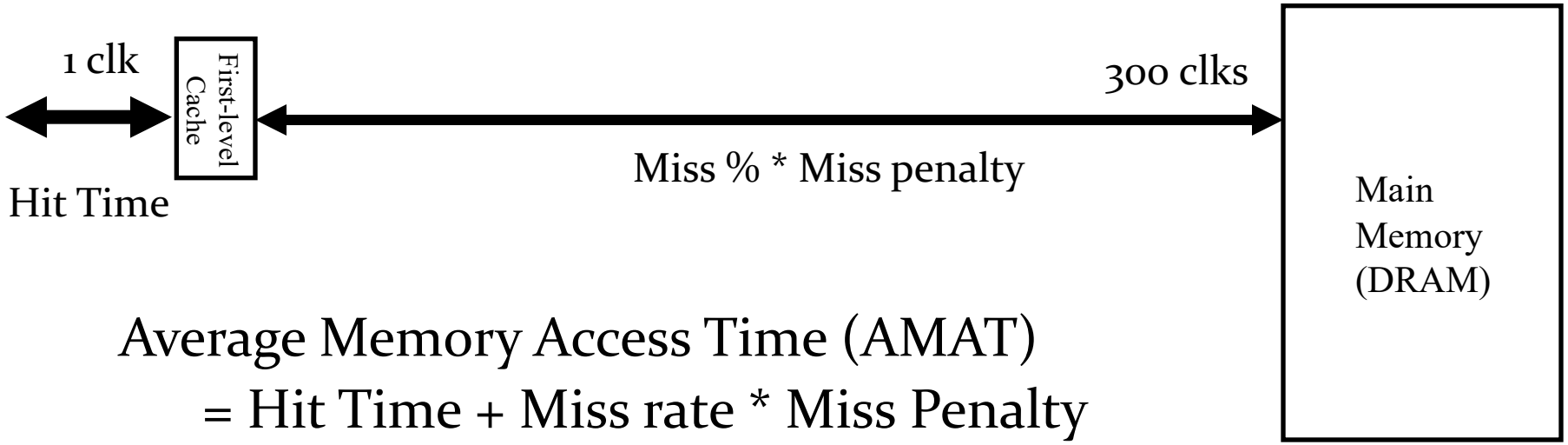
= Hit Time + Miss rate * Miss Penalty

= 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

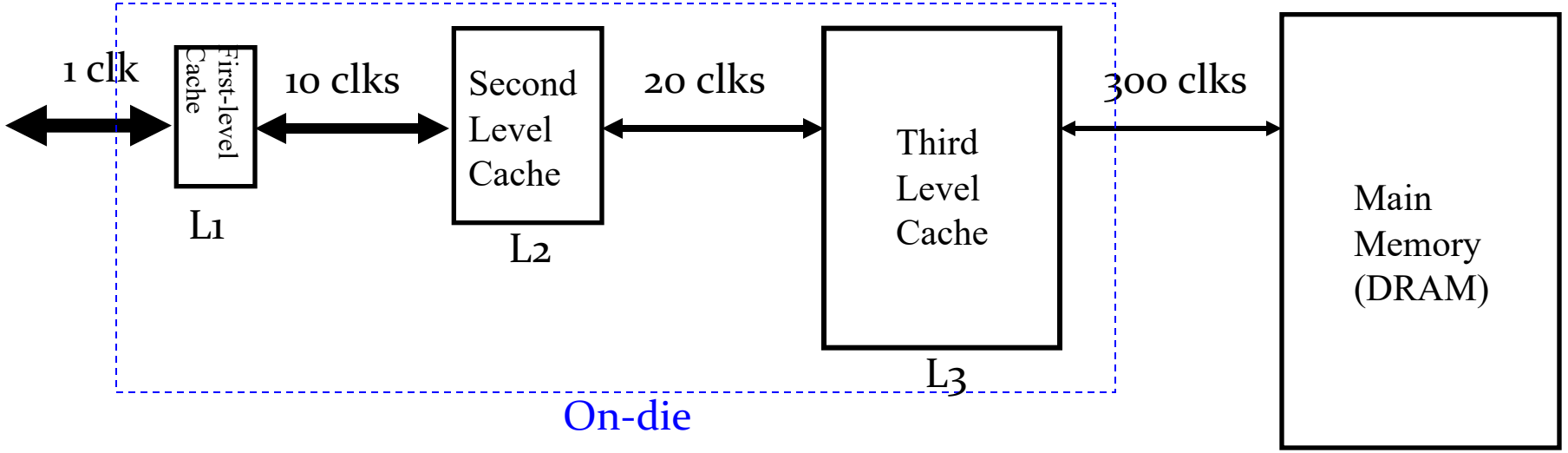
= T_{hit}(L1) + Miss%(L1) * T(memory)

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)

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

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 **any 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)

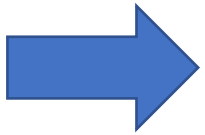
- DM and FA can be thought as special cases of SA
  - DM → 1-way SA
  - FA → All-way SA

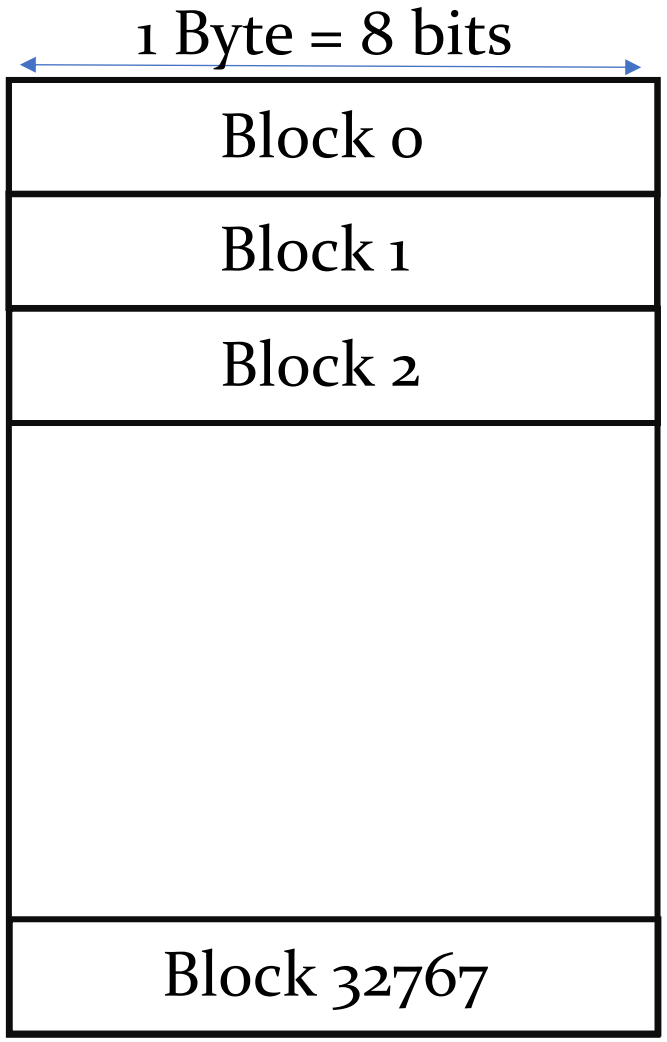
Direct Mapped Cache example

Assume 32 KByte cache with a 1 Byte cache block

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

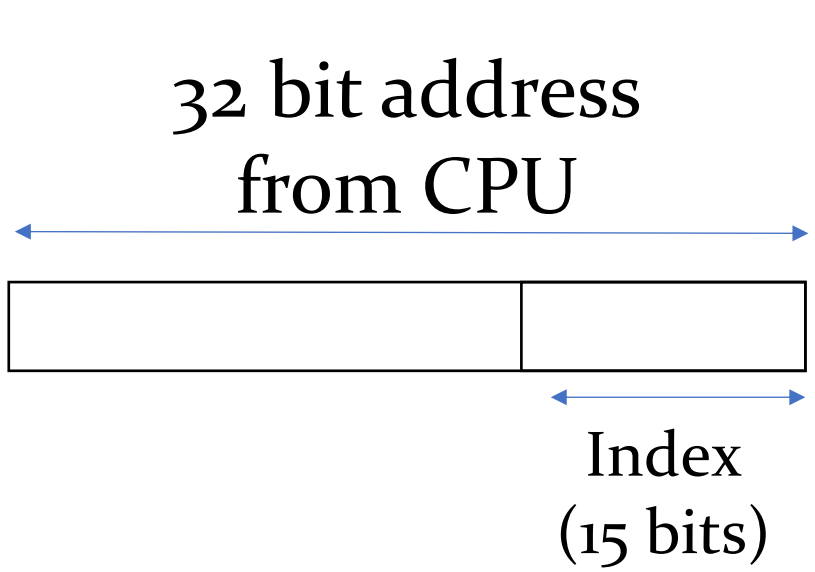
32 bit address from CPU



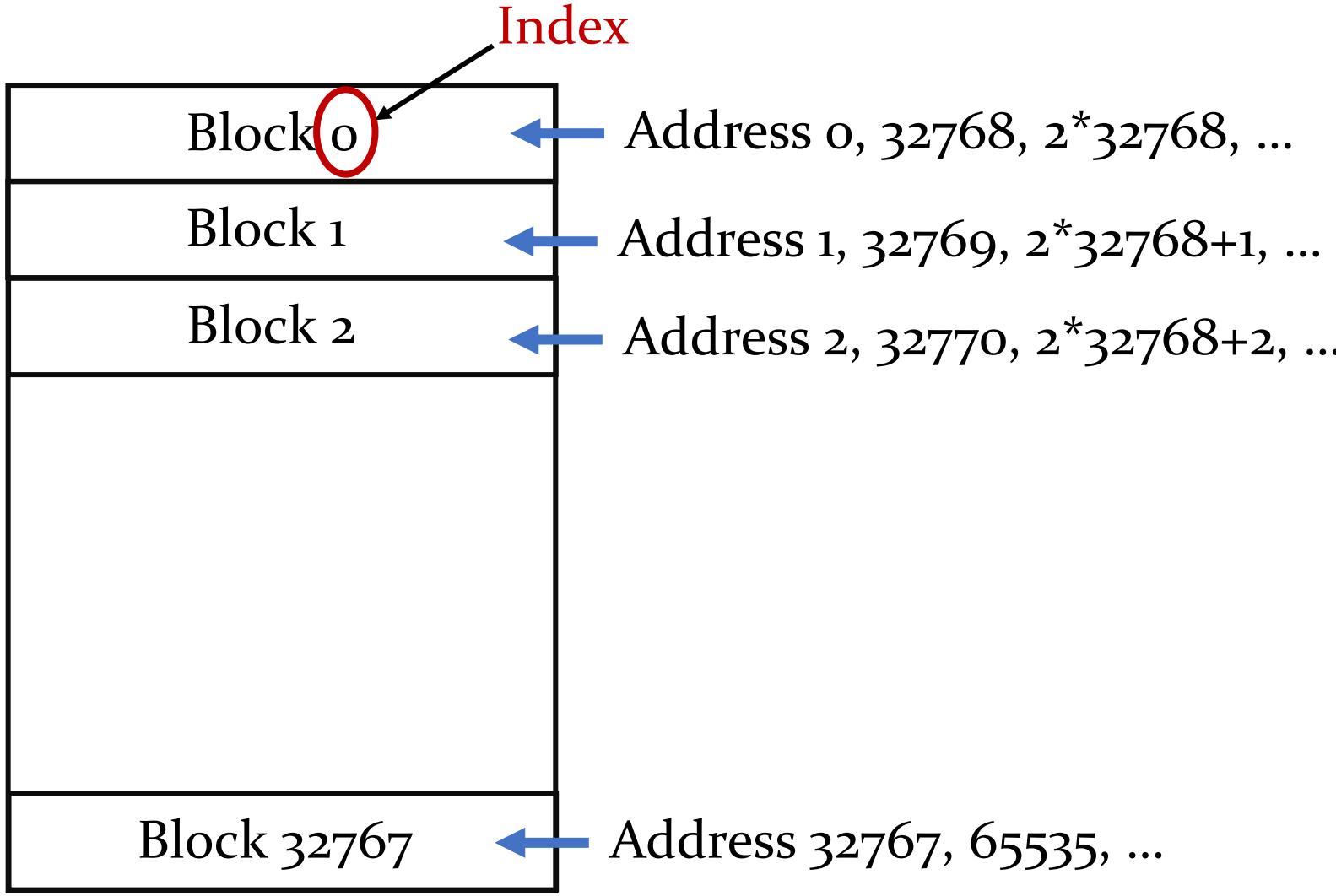


Direct Mapped Cache: Placement

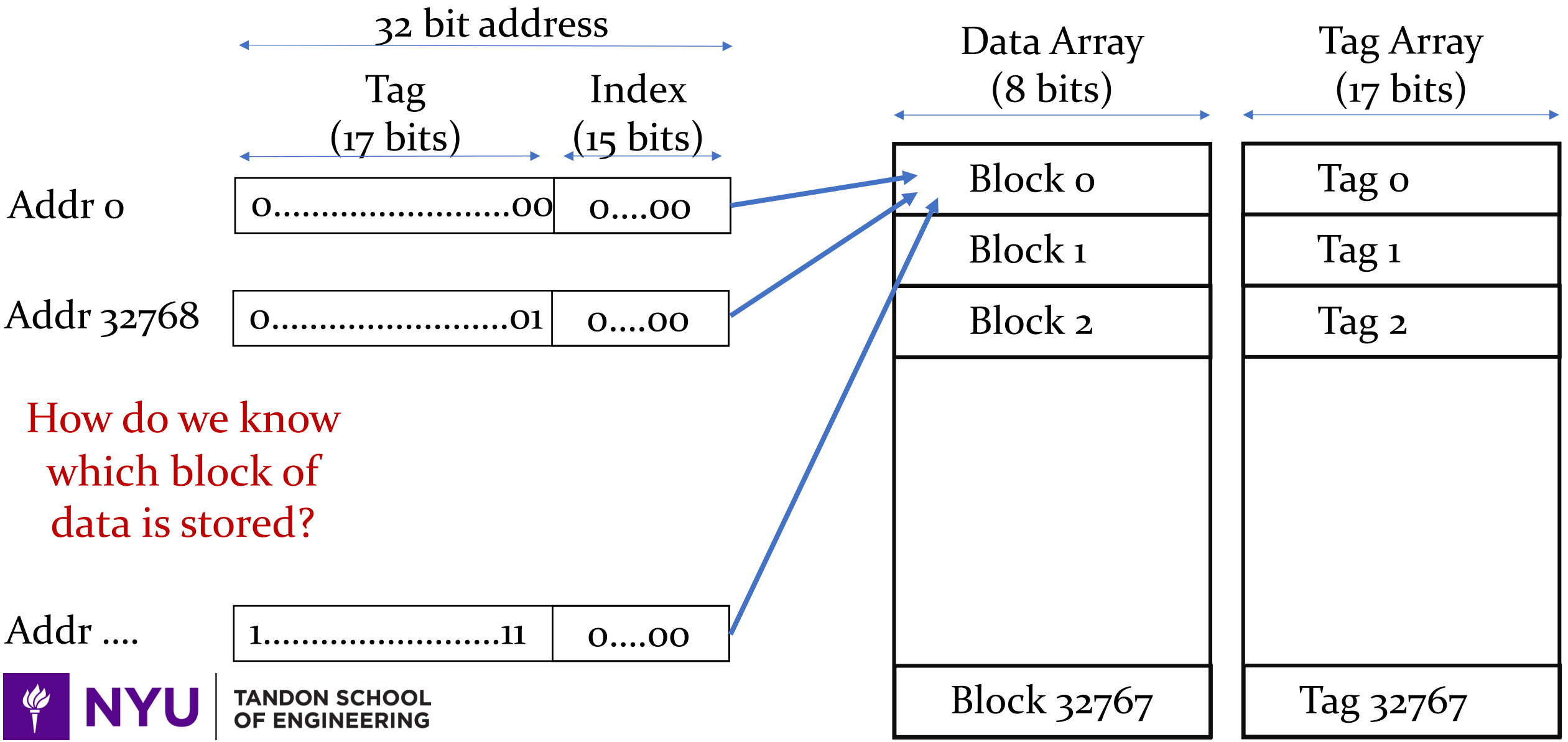
- Each block in the cache has a $\log_2(32768) = 15$ bit "index"



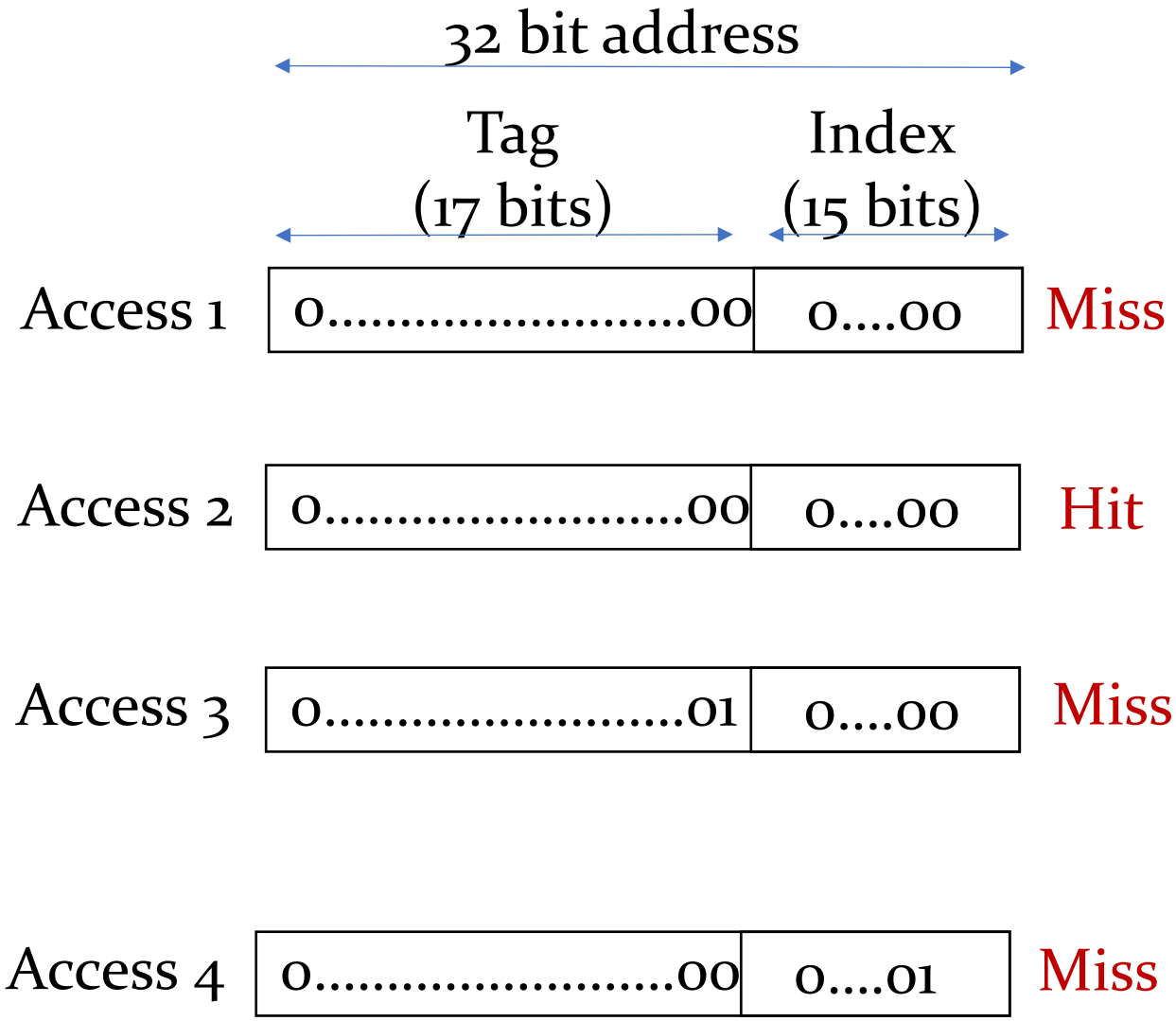
15 LSBs of address (index) used to look up/place block in cache

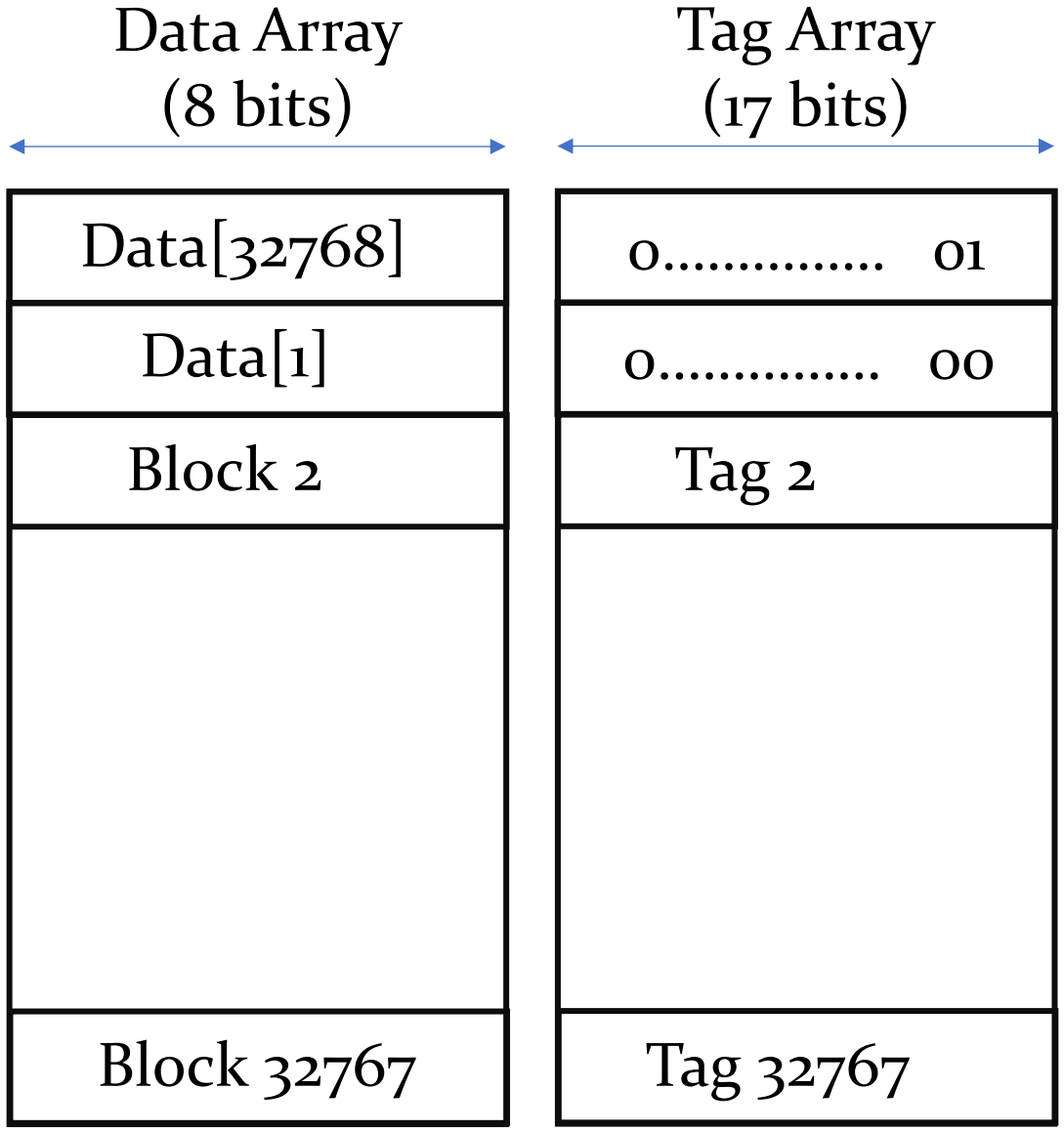


Direct Mapped Cache: Identification

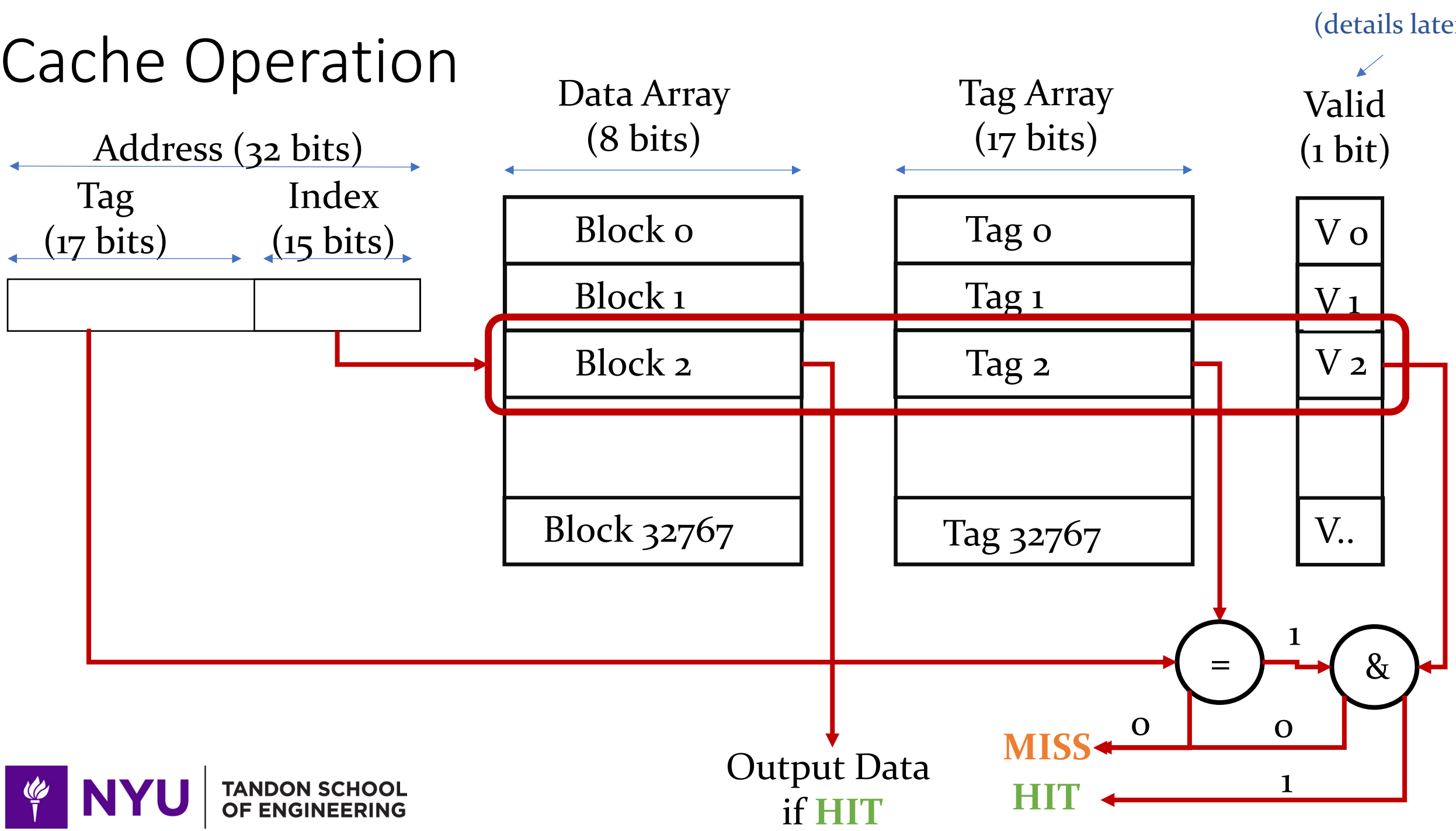


Direct Mapped Cache: Replacement





Cache Operation

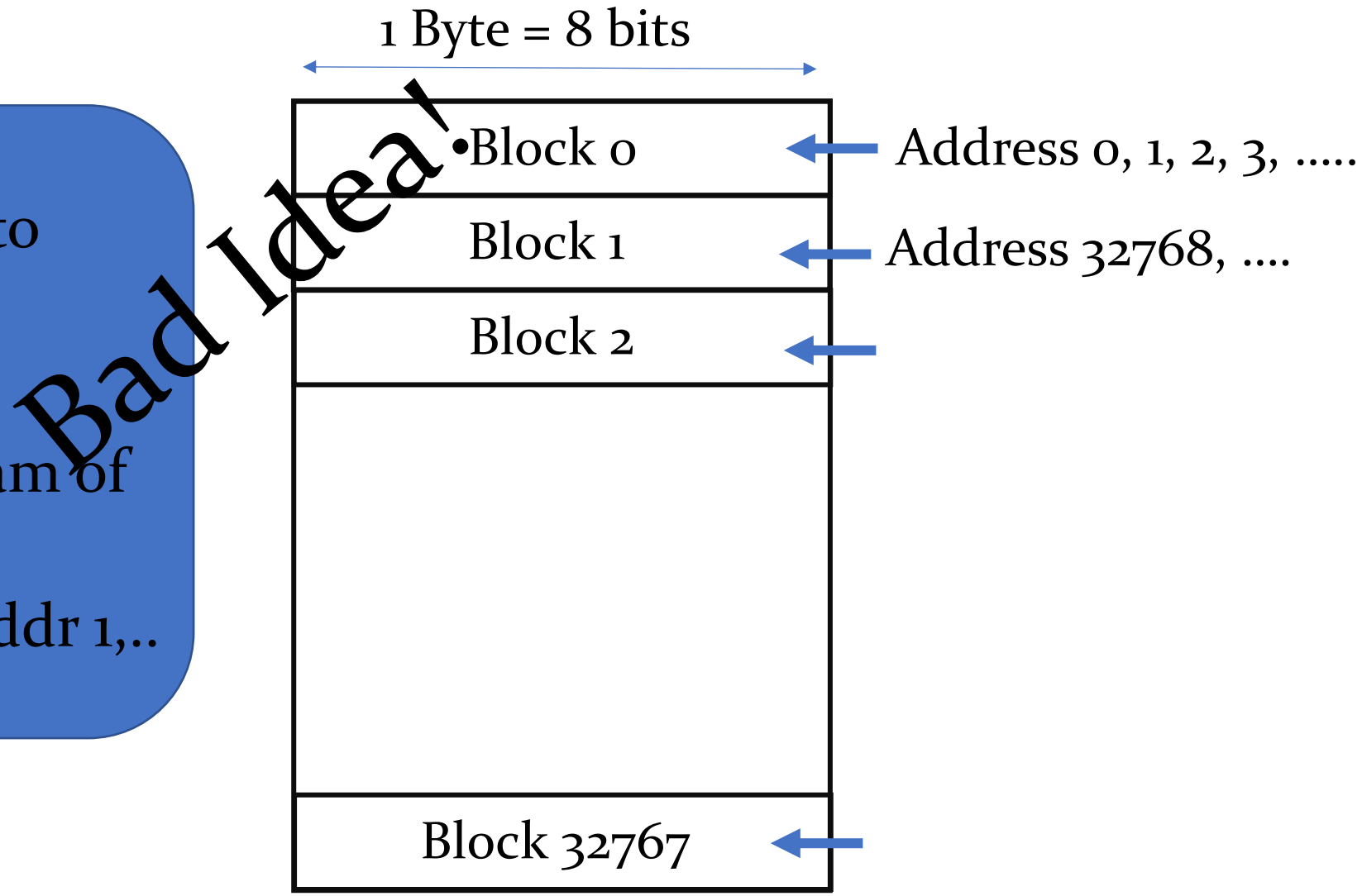


Alternative DM Placement

Assume 32 KByte cache with a 1 Byte cache block

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

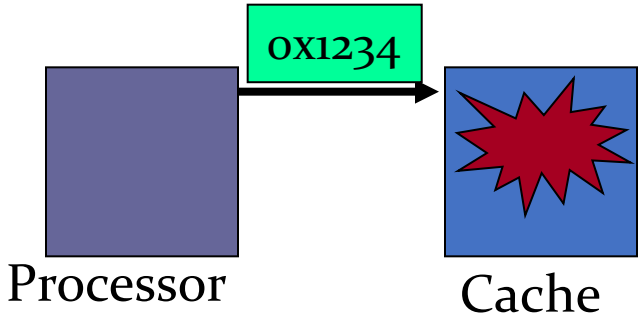
Addr 0, Addr 1, Addr 0, Addr 1,..



Three Cs (Cache Miss Terms)

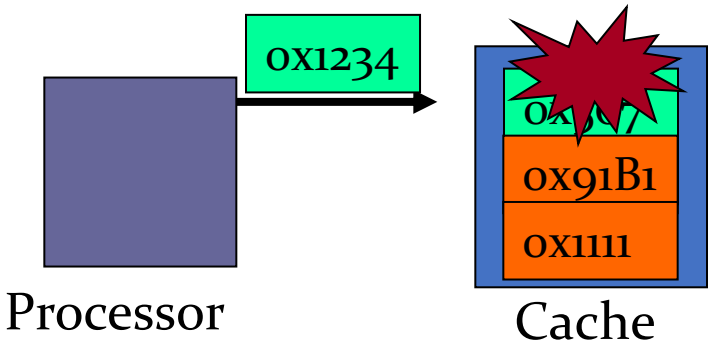
Compulsory Misses:

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



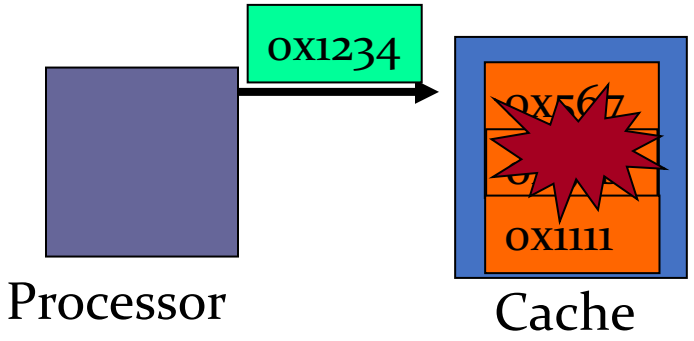
Conflict Misses:

- Increase cache size or associativity (next)
- 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

Assume addresses A and B with same index bits

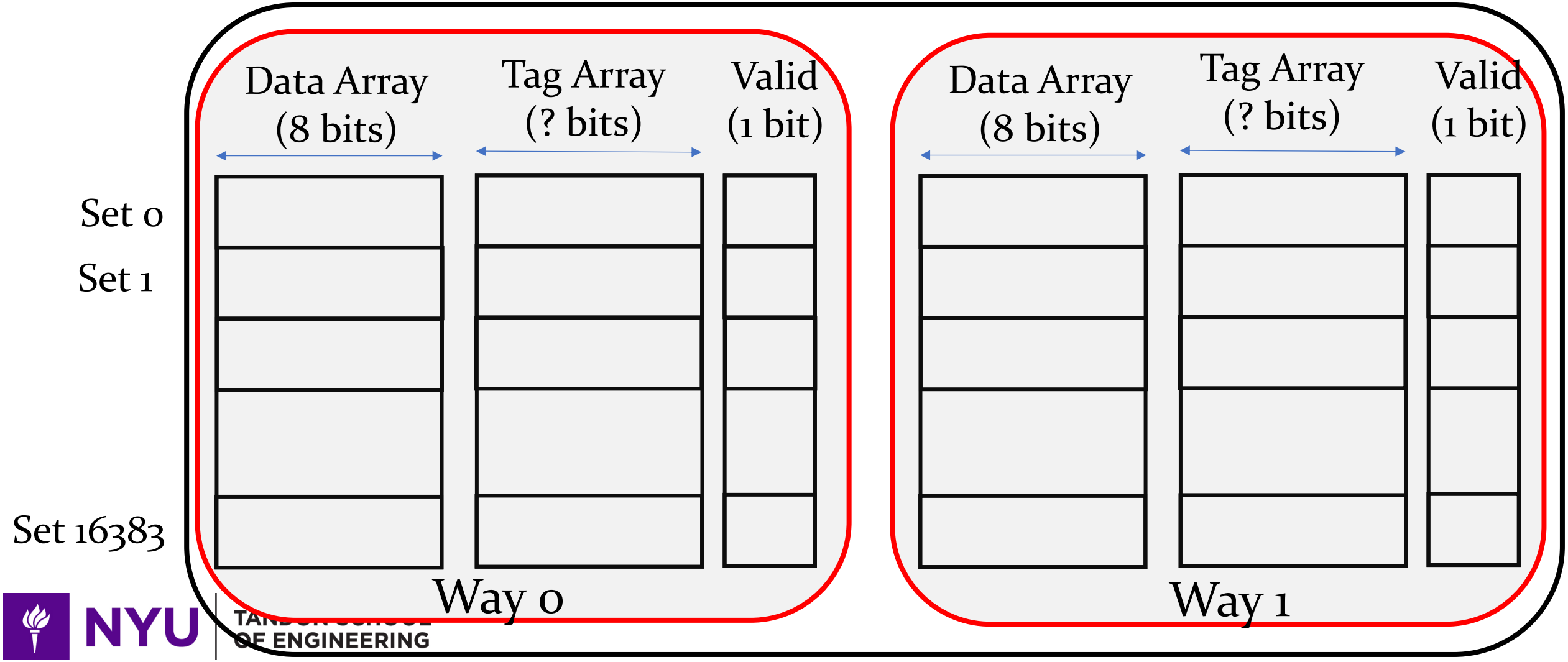
- Sequence: A, B, A, B, A….. results in 100% cache miss rate!

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

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

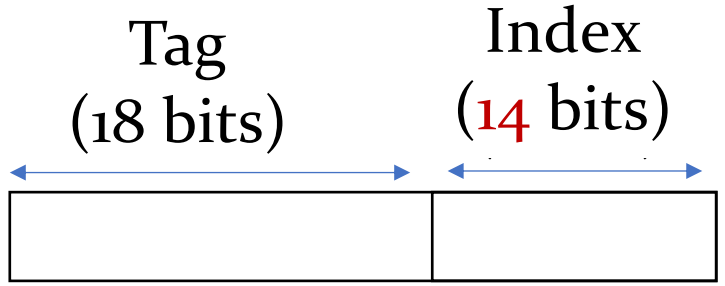
2-Way Set Associative Cache

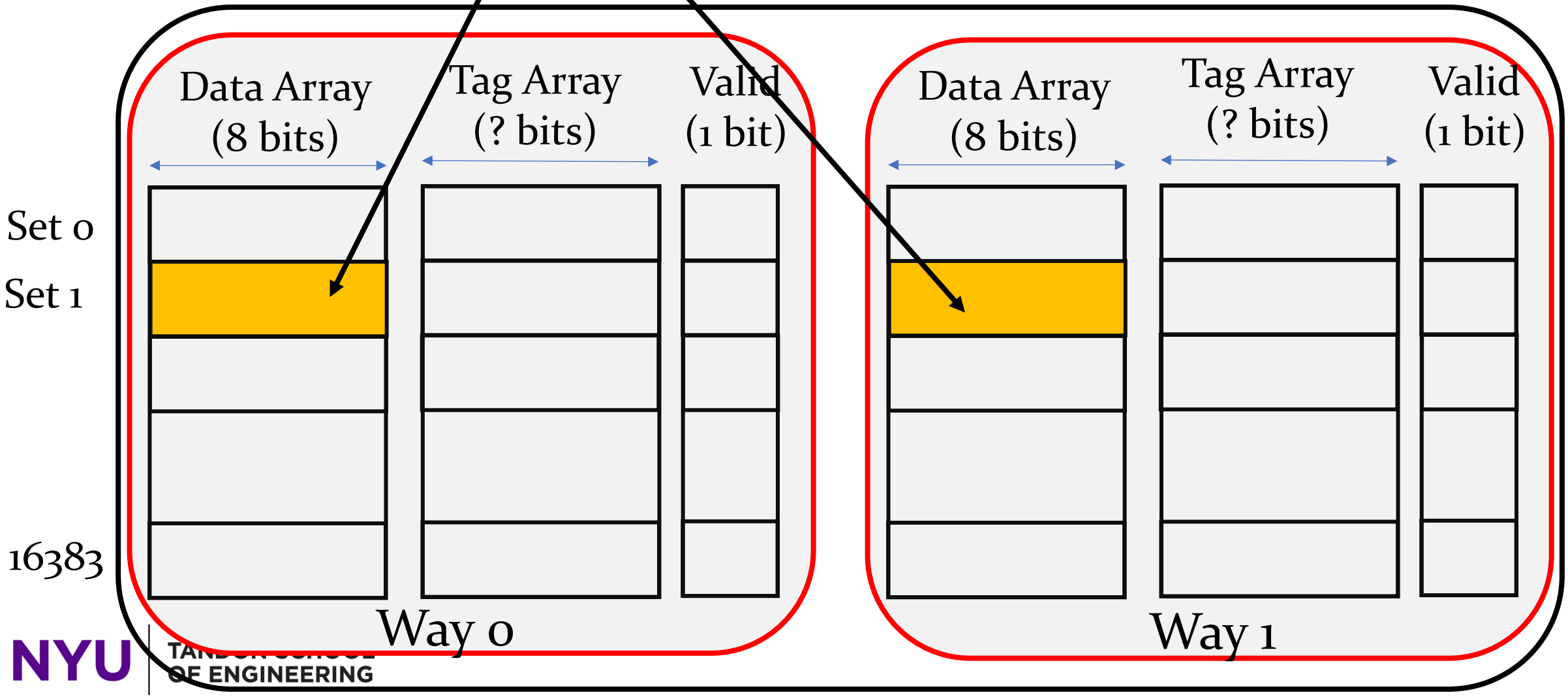
2-way set-associative 32 KB cache with 1 Byte blocks



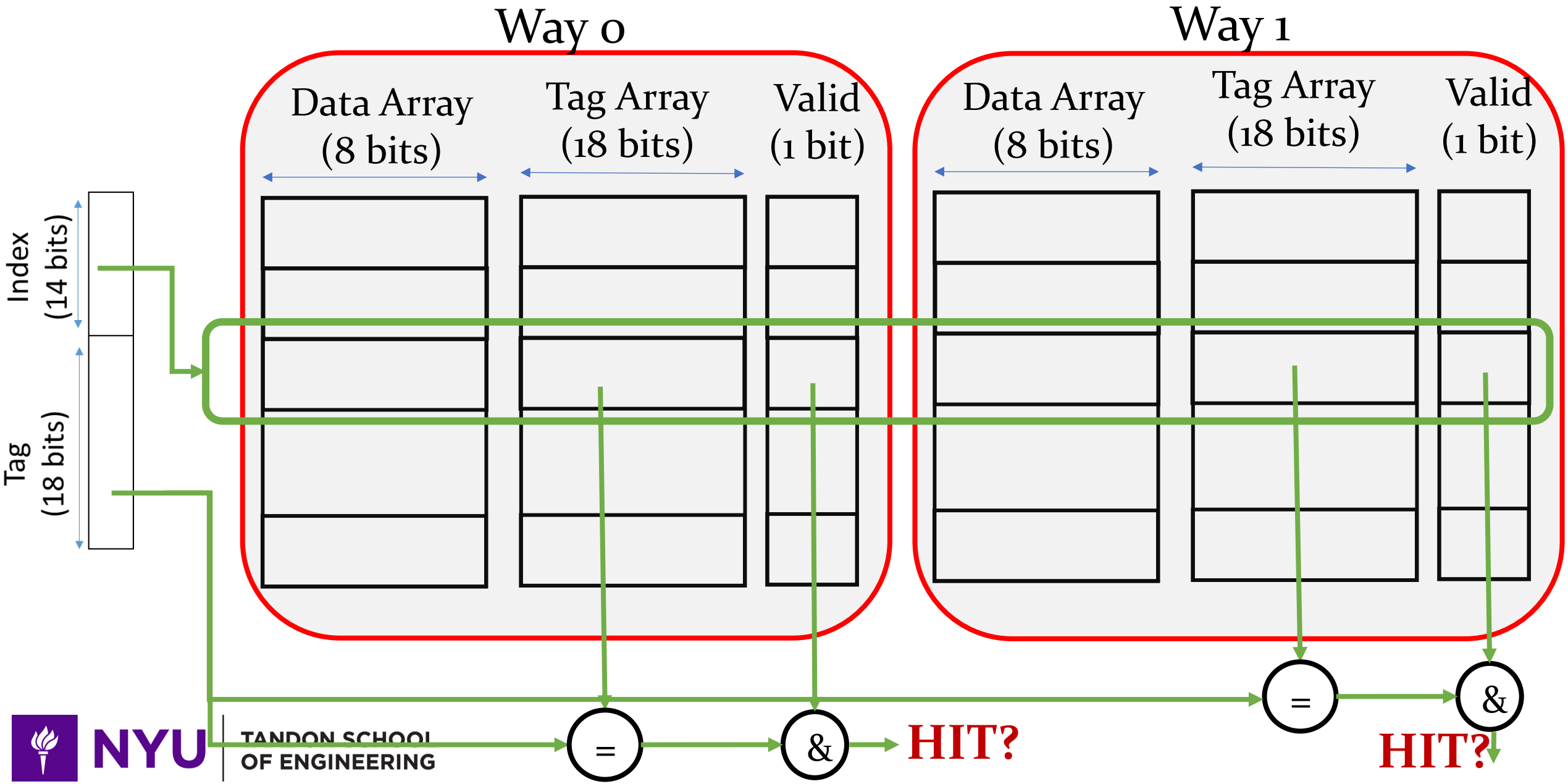
Placement

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



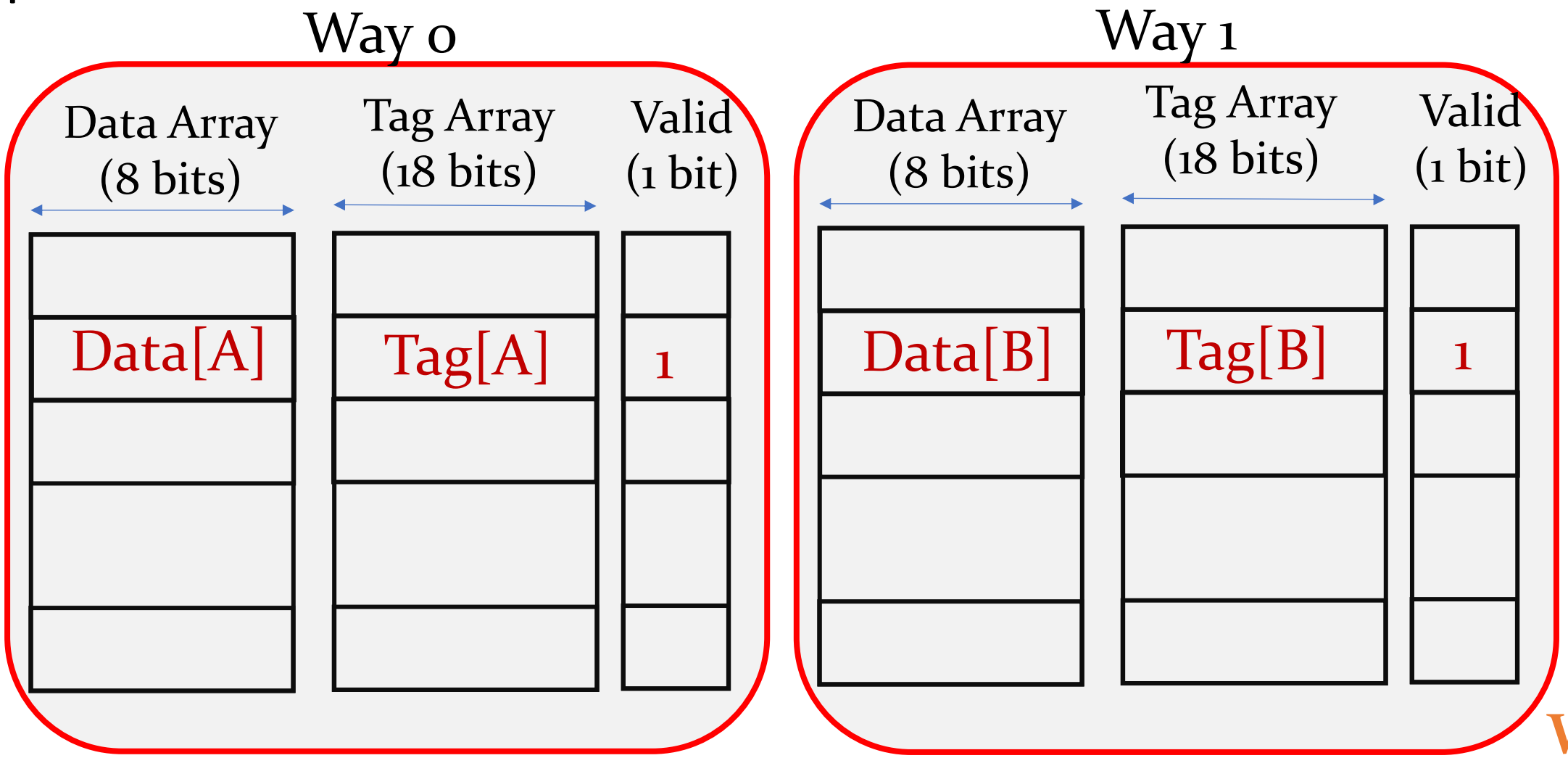


Identification



Replacement

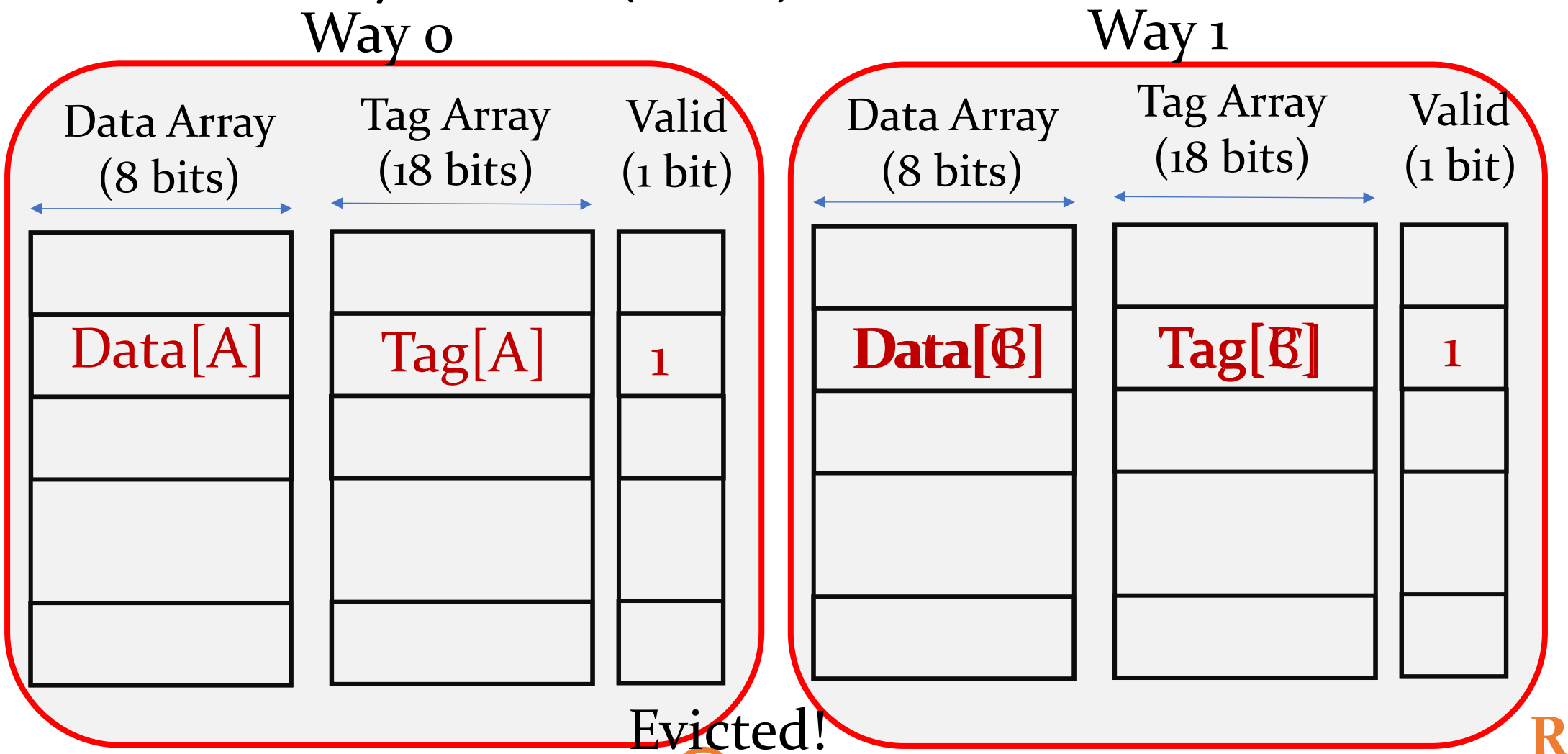
(Addresses A, B, C map to same set)



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

Where should C go?

Least Recently Used (LRU)

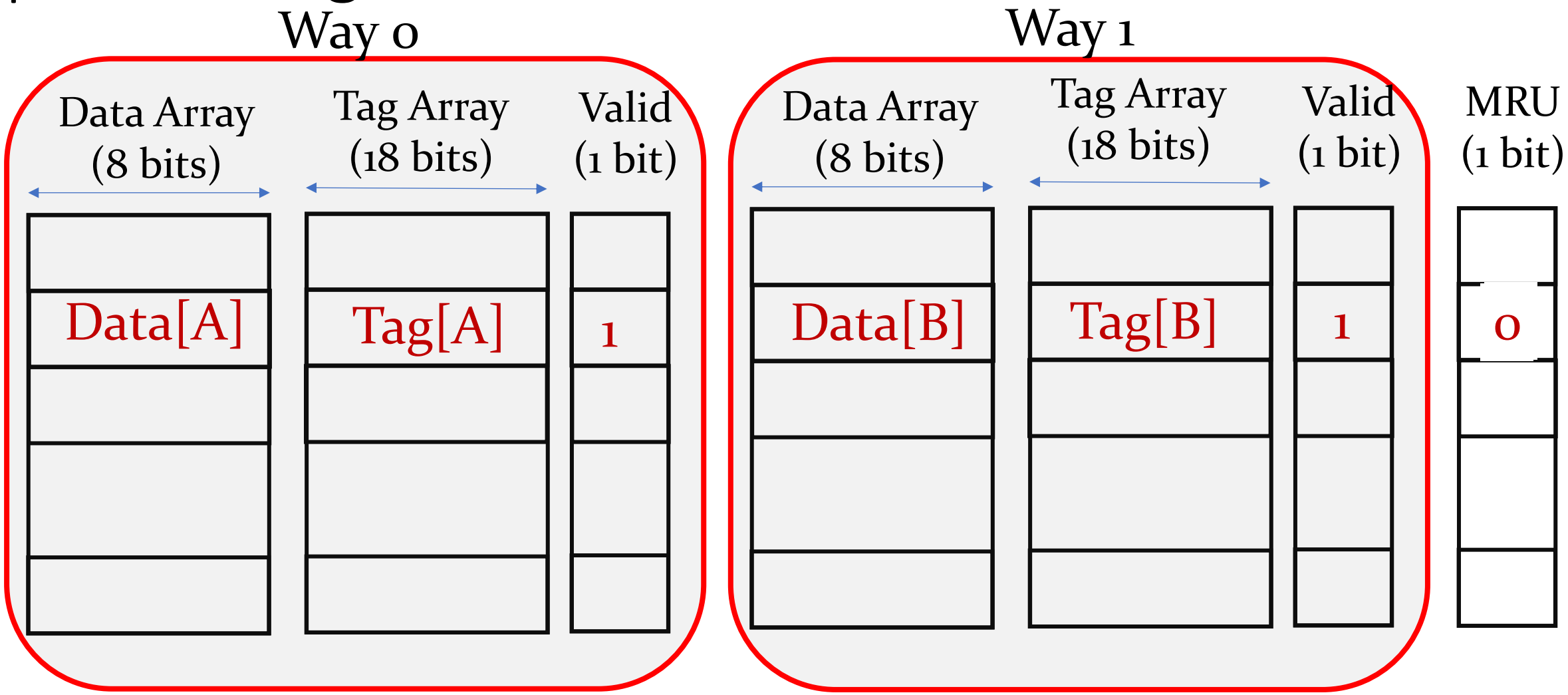


Evicted!

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

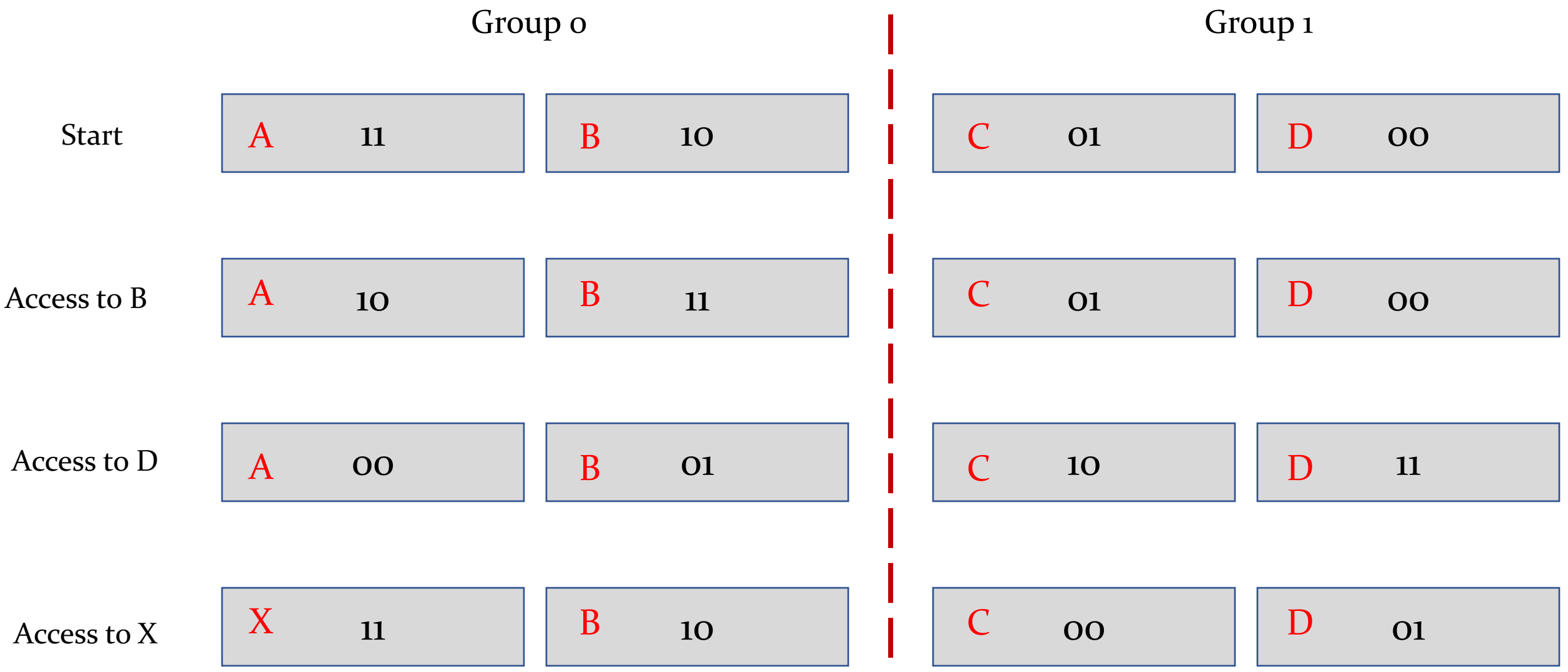
- Keep track of all possible N! orderings of N ways
- A linked list in which the head points to MRU and tail points to LRU
- Example: N=4; 4 x 2 bits = 8 bits per cache set and extra logic to update list on every access

Intuition: LRU is an approximation anyways..

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?

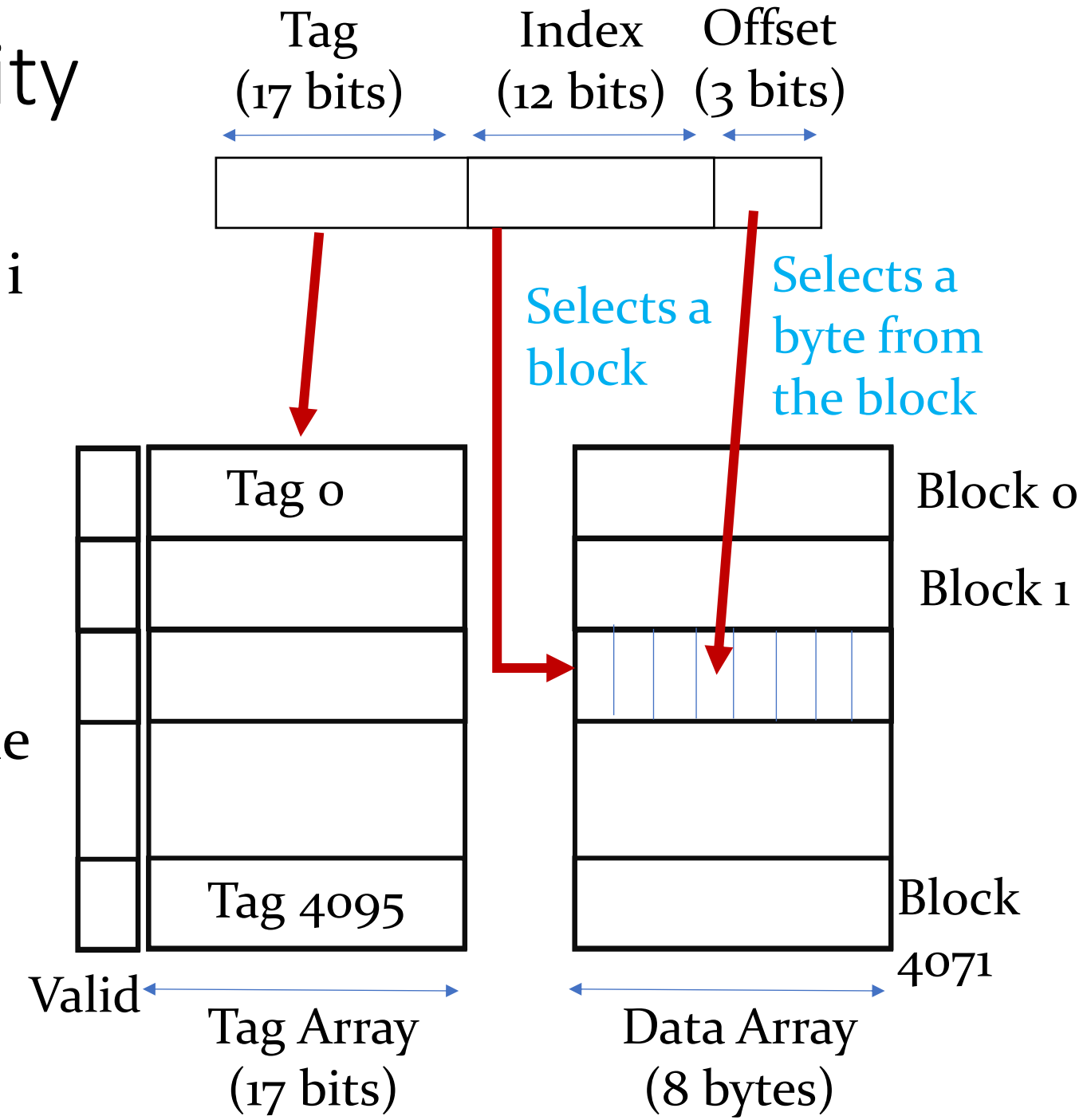
Exploiting Spatial Locality

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

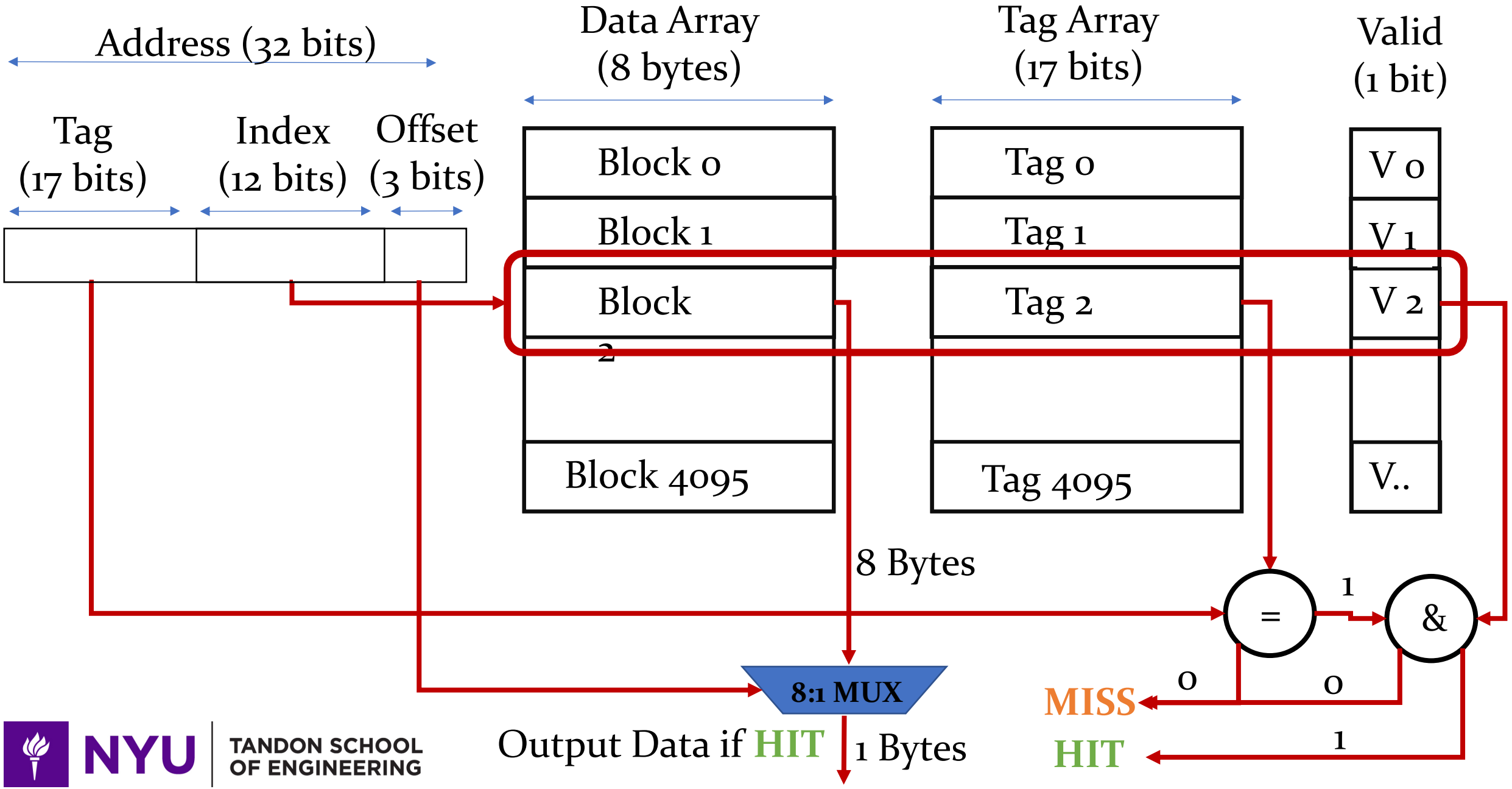
- Pull in *multiple* contiguous bytes of data in each access
- Use larger block size!

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

- i.e., cache has 4096 Byte blocks



Cache Operation



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