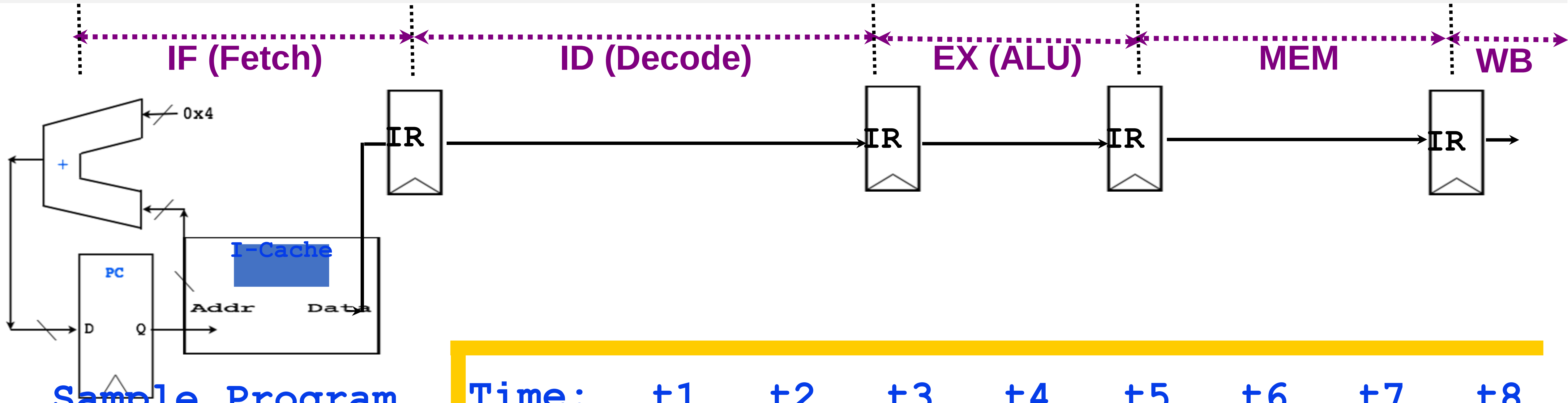
# World of Predictors: Branch/Value Predictors CASS ‘18

Biswa@CSE-IITK

image courtesy: stan’s world

# Basics First



**Sample Program (ISA w/o branch delay slot)**

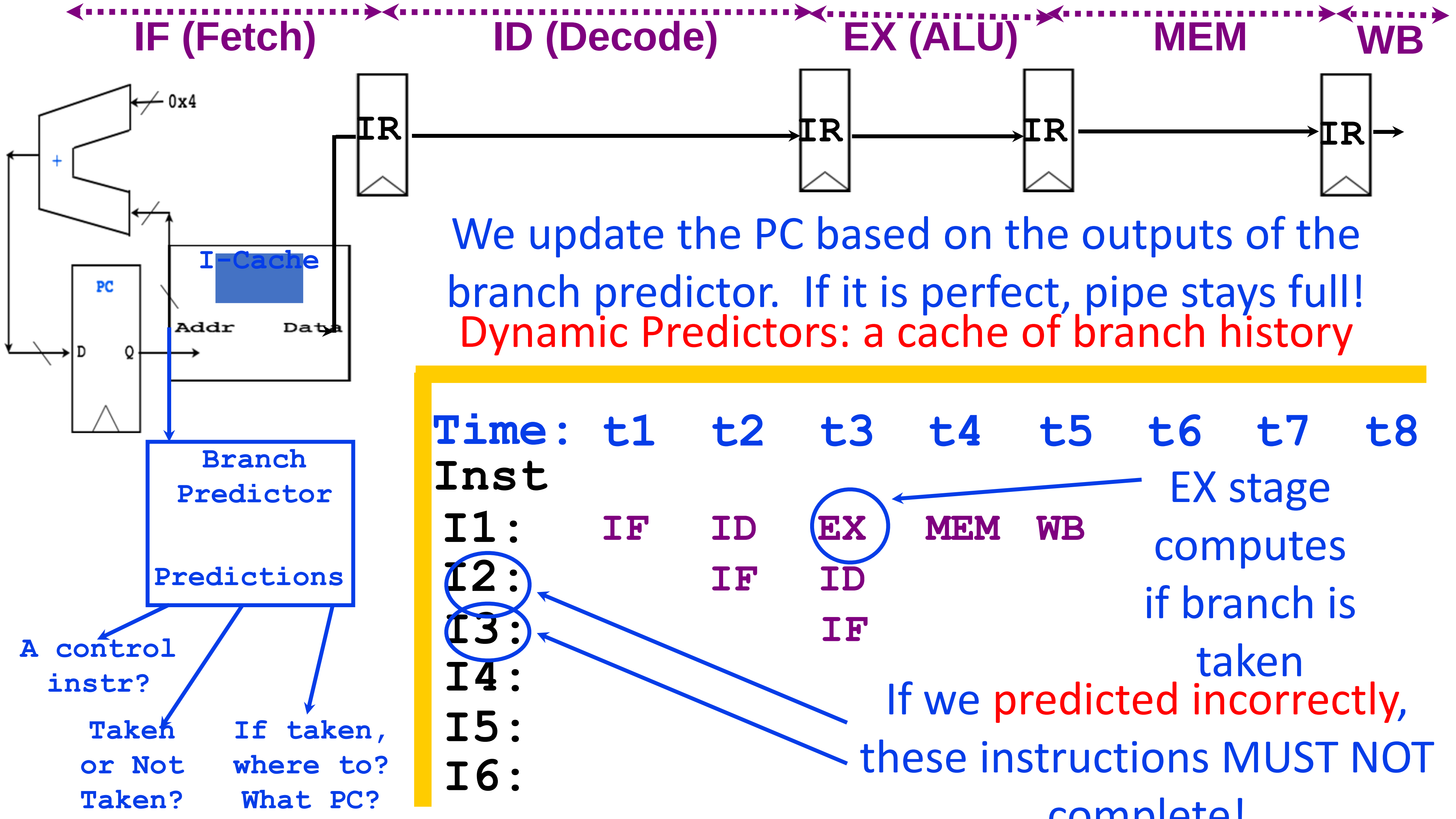
```
I1:   BEQ R4,R3,25
I2:   AND R6,R5,R4
I3:   SUB R1,R9,R8
```

| Time: | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 |
|---|---|---|---|---|---|---|---|---|
| Inst | | | | | | | | |
| I1: | IF | ID | EX | MEM | WB | | | |
| I2: | | IF | ID | | | | | |
| I3: | | | IF | | | | | |
| I4: | | | | | | | | |
| I5: | | | | | | | | |
| I6: | | | | | | | | |

EX stage computes if branch is taken

If branch is taken, these instructions MUST NOT complete!

# Welcome to Branch Prediction



We update the PC based on the outputs of the branch predictor. If it is perfect, pipe stays full!

Dynamic Predictors: a cache of branch history

| Time: | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 |
|---|---|---|---|---|---|---|---|---|
| Inst | | | | | | | | |
| I1: | IF | ID | EX | MEM | WB | | | |
| I2: | | IF | ID | | | | | |
| I3: | | | IF | | | | | |
| I4: | | | | | | | | |
| I5: | | | | | | | | |
| I6: | | | | | | | | |

EX stage computes if branch is taken

If we predicted incorrectly, these instructions MUST NOT complete!

## Branch Prediction

- Idea: Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

# Static Branch Prediction

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

# Static Branch Prediction

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy

TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

# Dynamic Branch Prediction

- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages

  + Prediction based on history of the execution of branches

    + It can adapt to dynamic changes in branch behavior

  + No need for static profiling: input set representativeness problem goes away

- Disadvantages

  -- More complex (requires additional hardware)

# Simplest One: Last-Time Predictor

- Last time predictor
  - Indicates which direction branch went last time it executed

    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

- Always mis-predicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

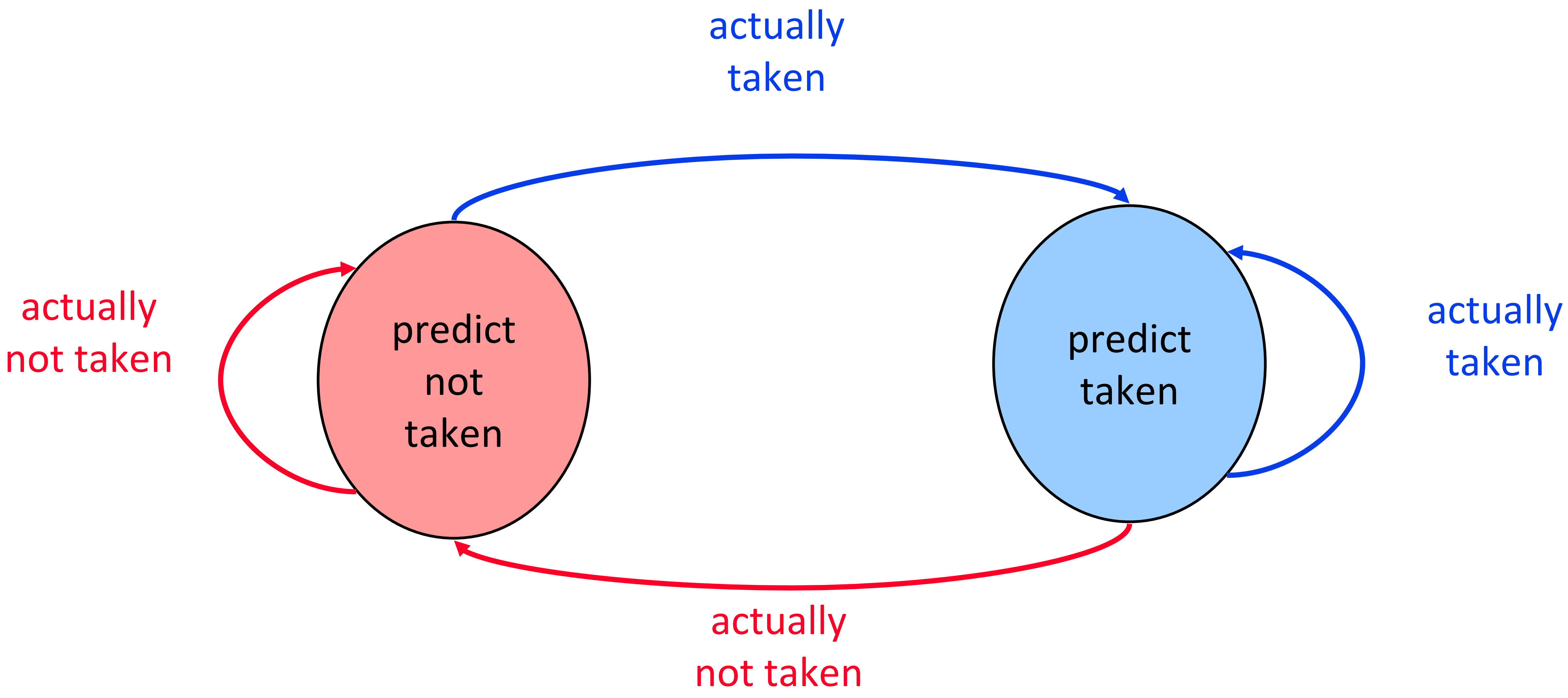
+ Loop branches for loops with large number of iterations

-- Loop branches for loops will small number of iterations

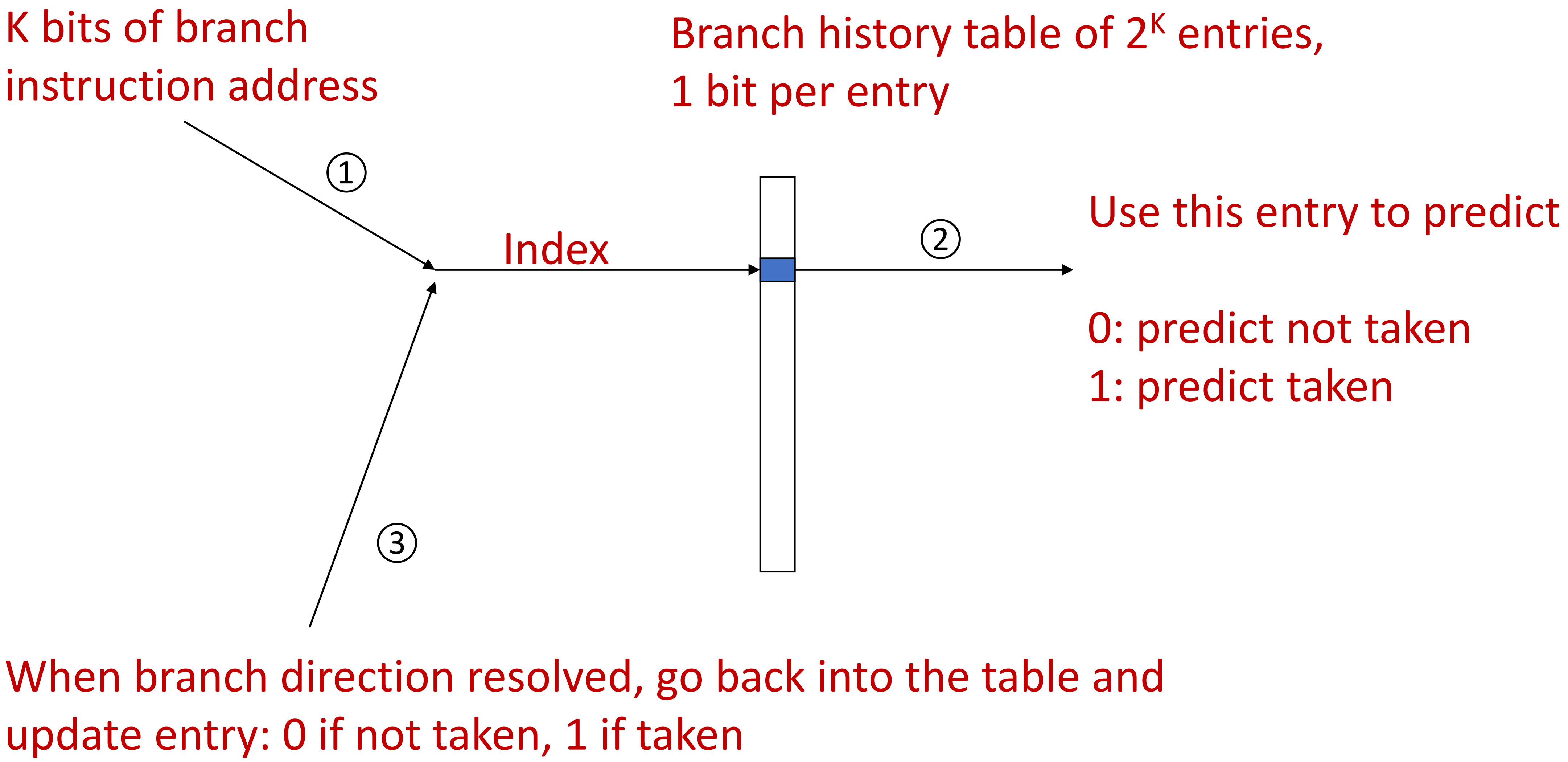
TNTNTNTNTNTNTNTNTNTN → 0% accuracy

*Last-time predictor CPI = [ 1 + (0.20*<u>0.15</u>) * 2 ] = 1.06 (Assuming 85% accuracy)*

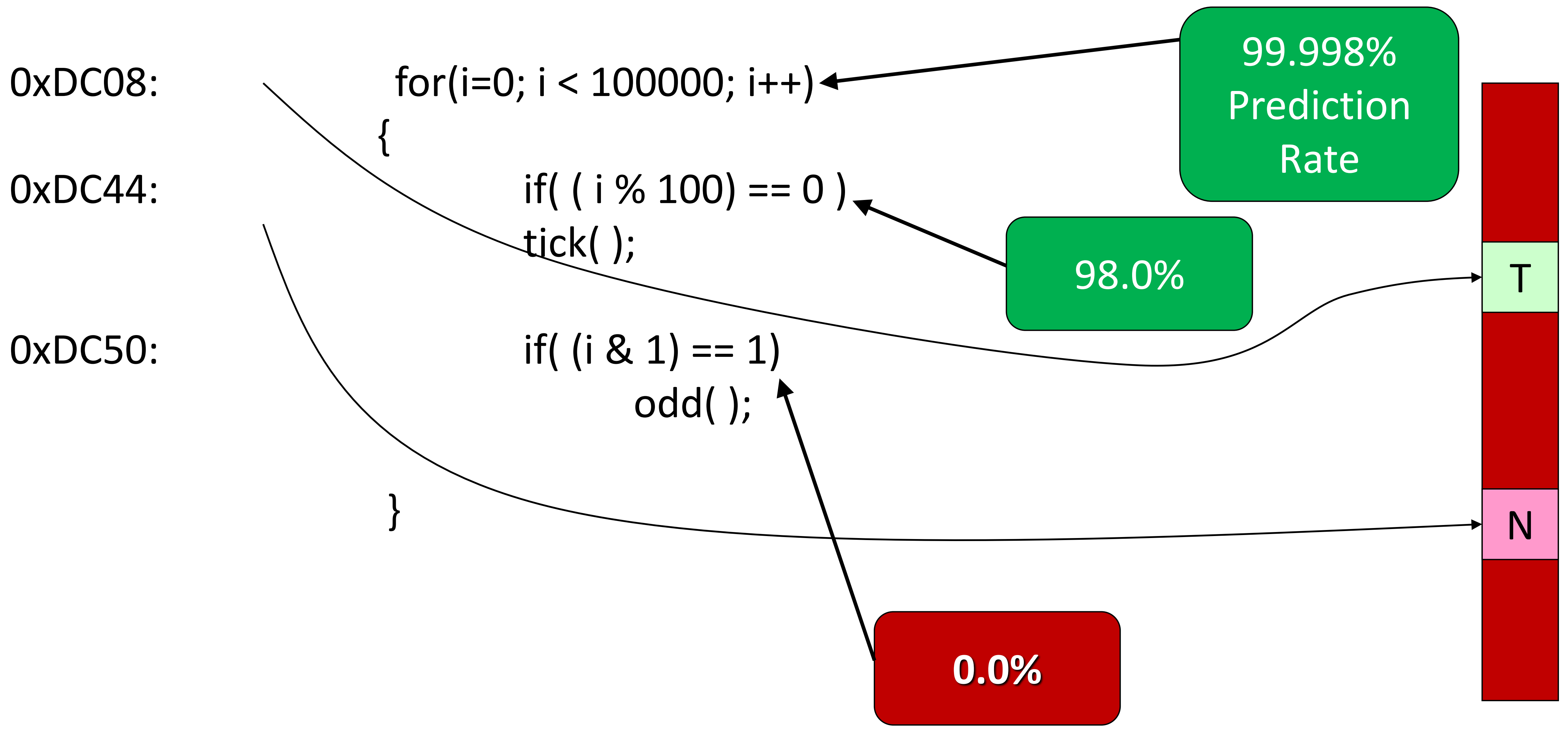
## Last-Time


actually taken
actually not taken
predict not taken
predict taken
actually taken
actually not taken

# Last-time Predictor: The hardware


K bits of branch instruction address
①
Index
Branch history table of $2^K$ entries, 1 bit per entry
②
Use this entry to predict
0: predict not taken
1: predict taken
③
When branch direction resolved, go back into the table and update entry: 0 if not taken, 1 if taken

# Example: Predict!!

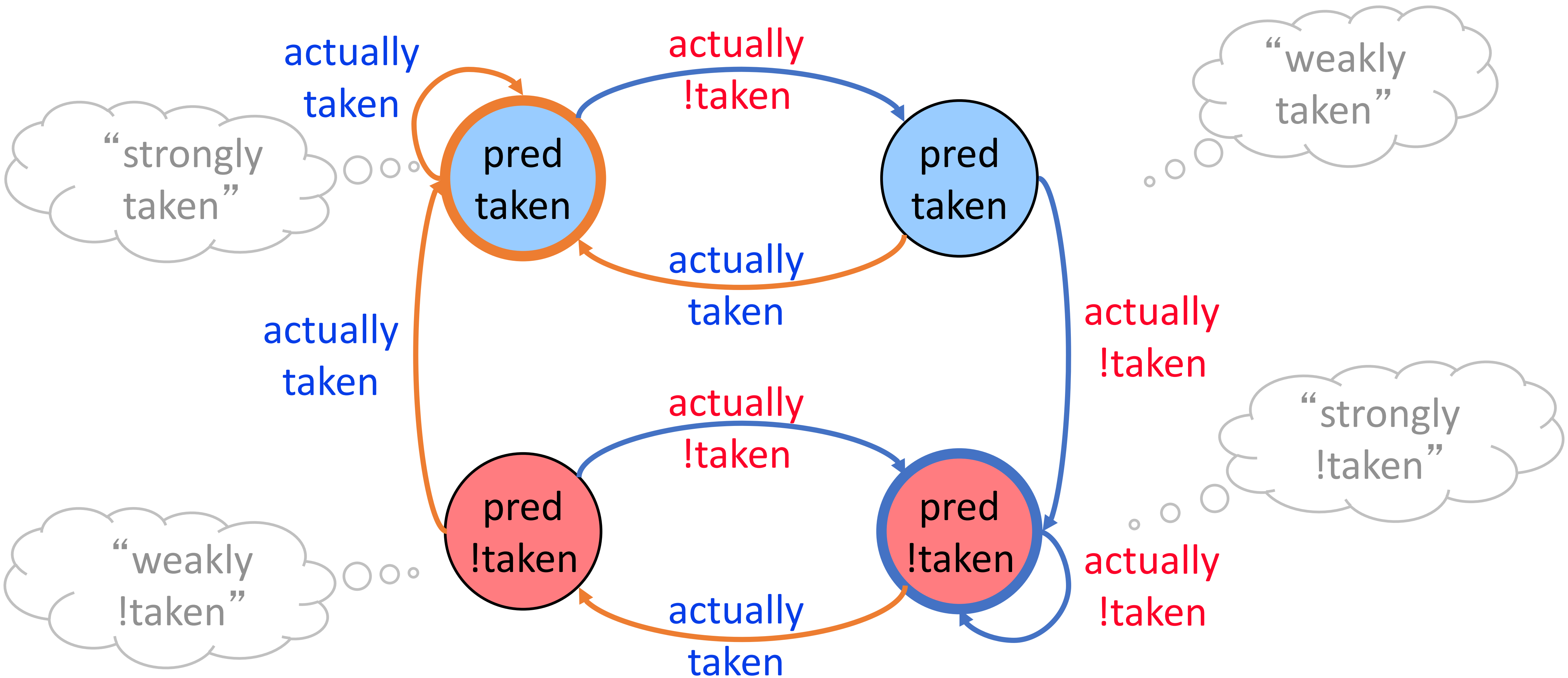


```
0xDC08:     for(i=0; i < 100000; i++)
            {
0xDC44:          if( ( i % 100) == 0 )
                 tick( );

0xDC50:          if( (i & 1) == 1)
                      odd( );

            }
```

# Change Predictor after 2 Mistakes



## Is This Enough

- Control flow instructions (branches) are frequent
    - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
    - N cycles: (minimum) branch resolution latency
    - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)

- How do we keep the pipeline full after a branch?

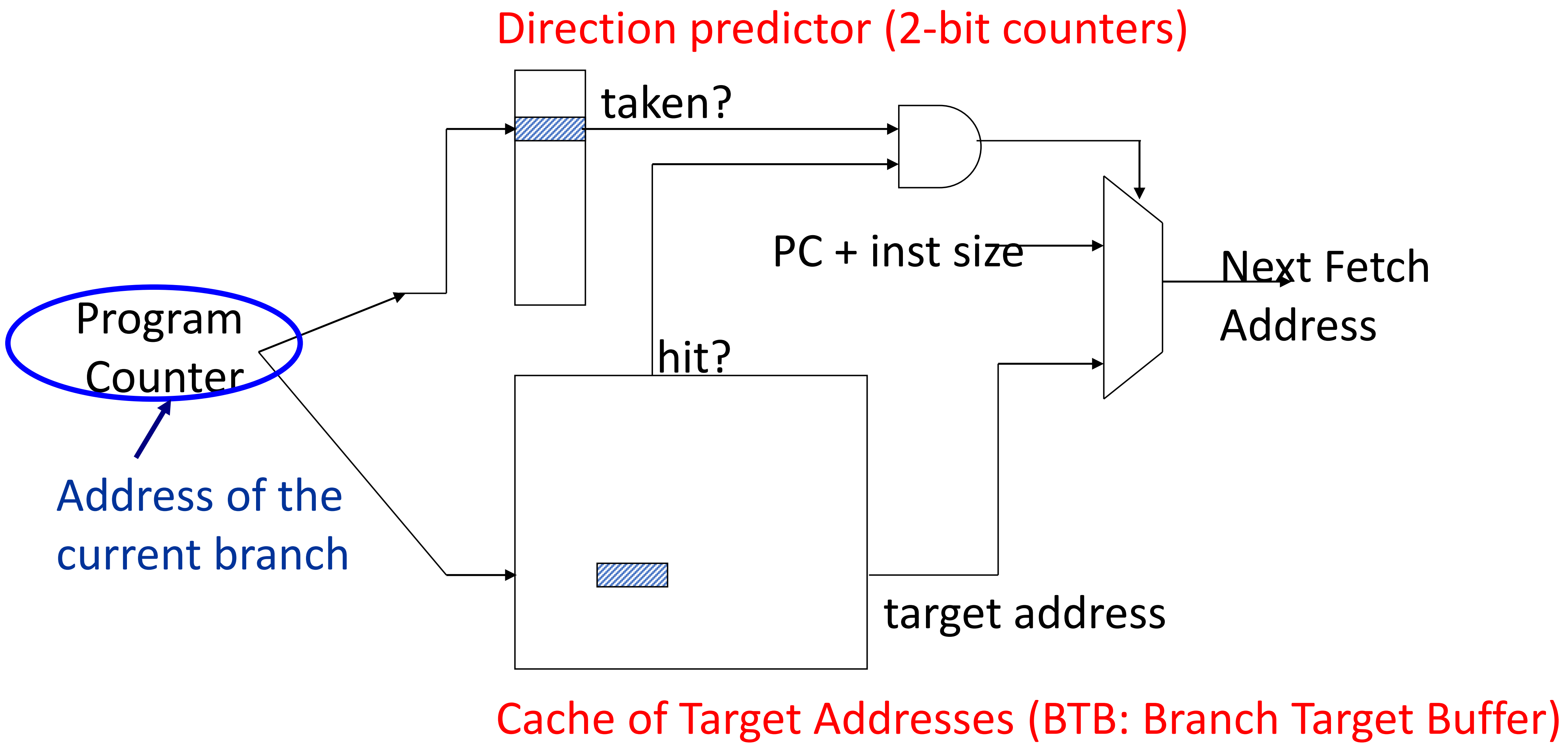
- Problem: Need to determine the **next fetch address** when the branch is fetched (to avoid a pipeline bubble)

## Is This Enough?

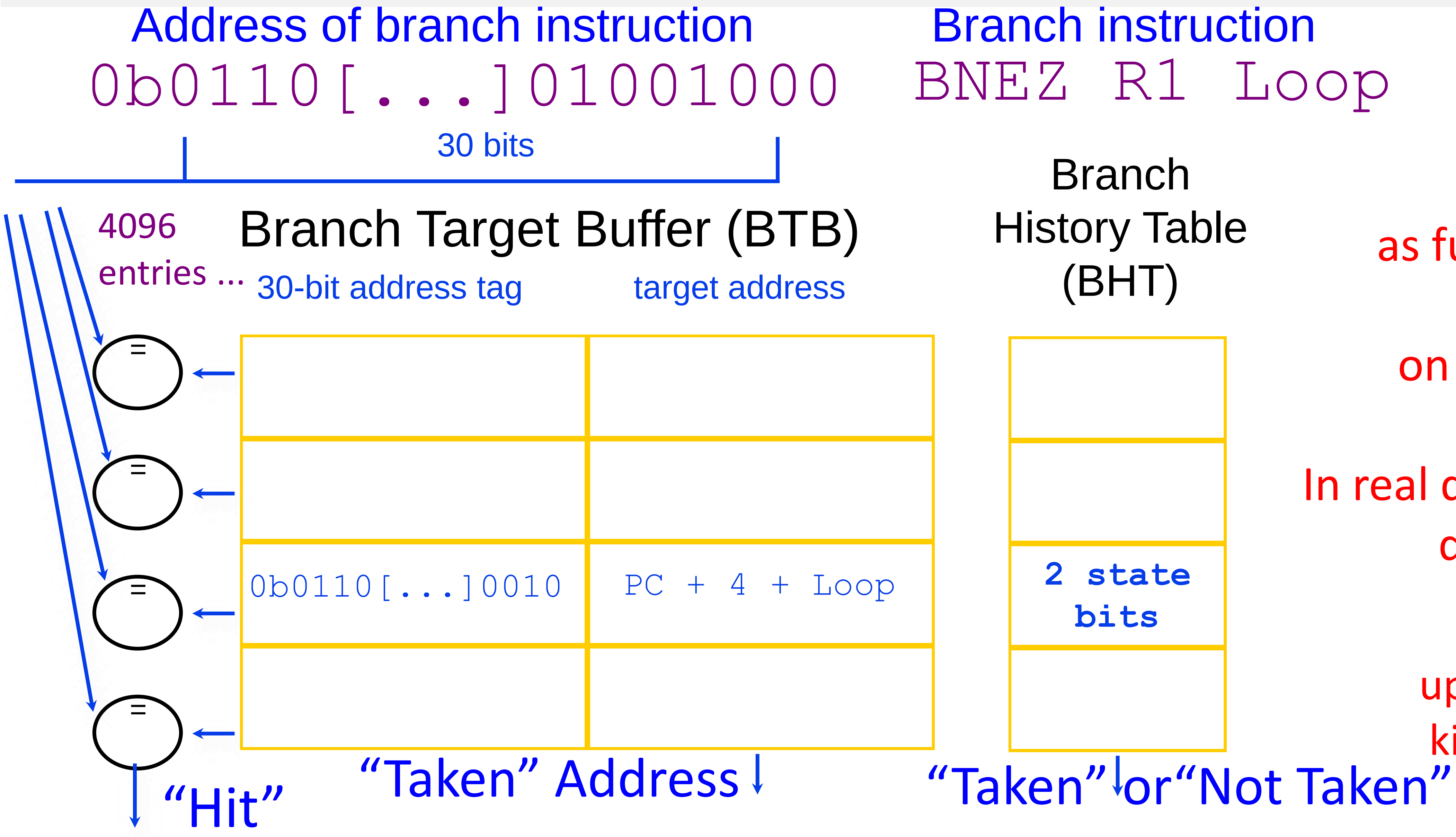
- Assume a pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 100 5-instruction blocks (500 instructions)?
  - Assume 1 out of 5 instructions is a branch, fetch width of 5, each 5 instruction block ends in a branch
  - 100% accuracy : 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy: 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy: 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy: 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched

# Fetch Stage with BTB and Direction Prediction


Direction predictor (2-bit counters)
taken?
PC + inst size
Next Fetch Address
hit?
Program Counter
Address of the current branch
target address
Cache of Target Addresses (BTB: Branch Target Buffer)

# BTB (Why 30-bit Tag?)

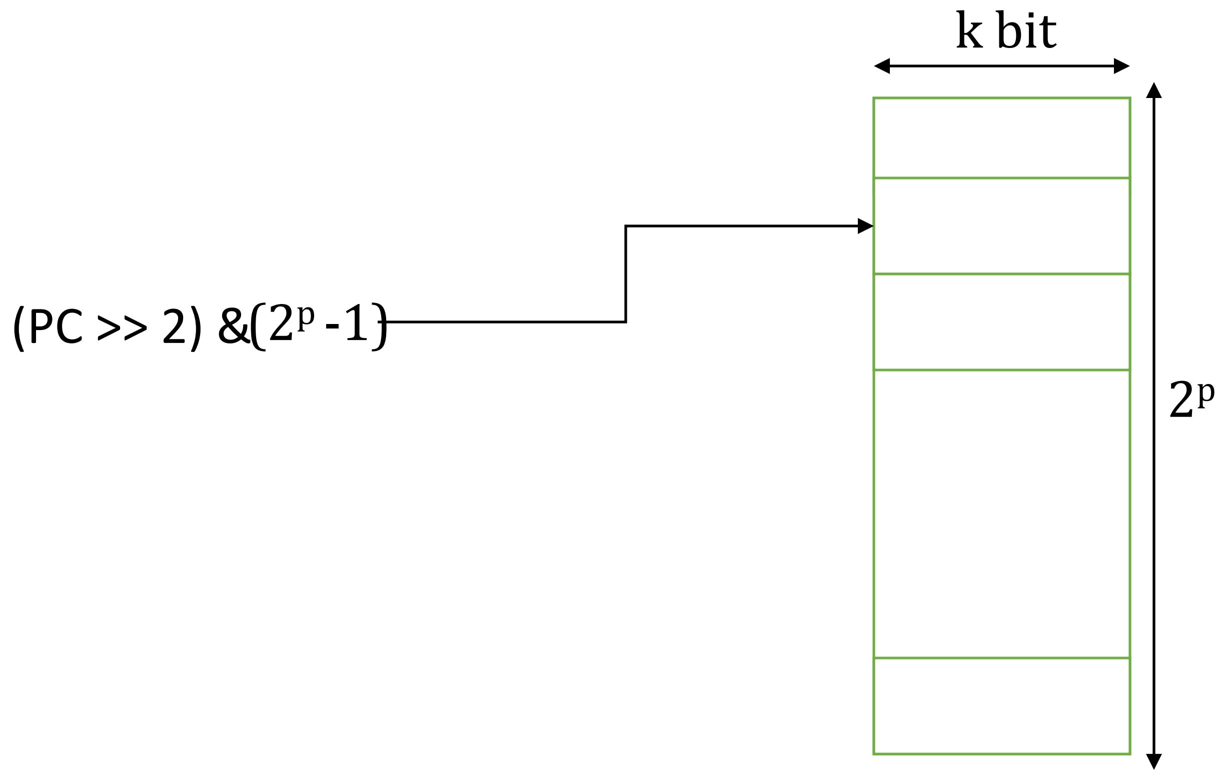


Drawn as fully associative to focus on the essentials.

In real designs, always direct-mapped.

At EX stage, update BTB/BHT, kill instructions, if necessary,

# No History based Branch Predictor



Bimodal predictor: Good for biased branches

## Local History & Global History

- Local Behavior

  What is the predicted direction of Branch A given the outcomes of previous instances of Branch A?

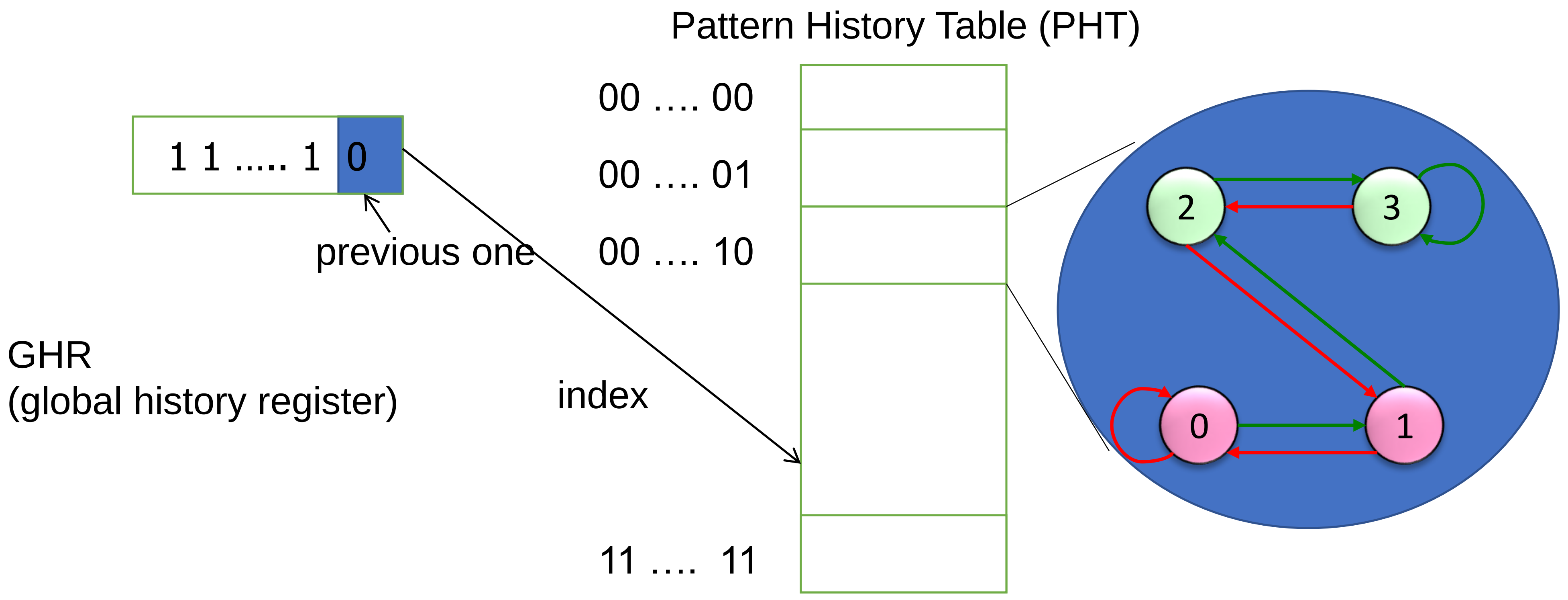
- Global Behavior

  What is the predicted direction of Branch Z given the outcomes of *all** previous branches A, B, …, X and Y?

  * Number of previous branches tracked limited by the history length

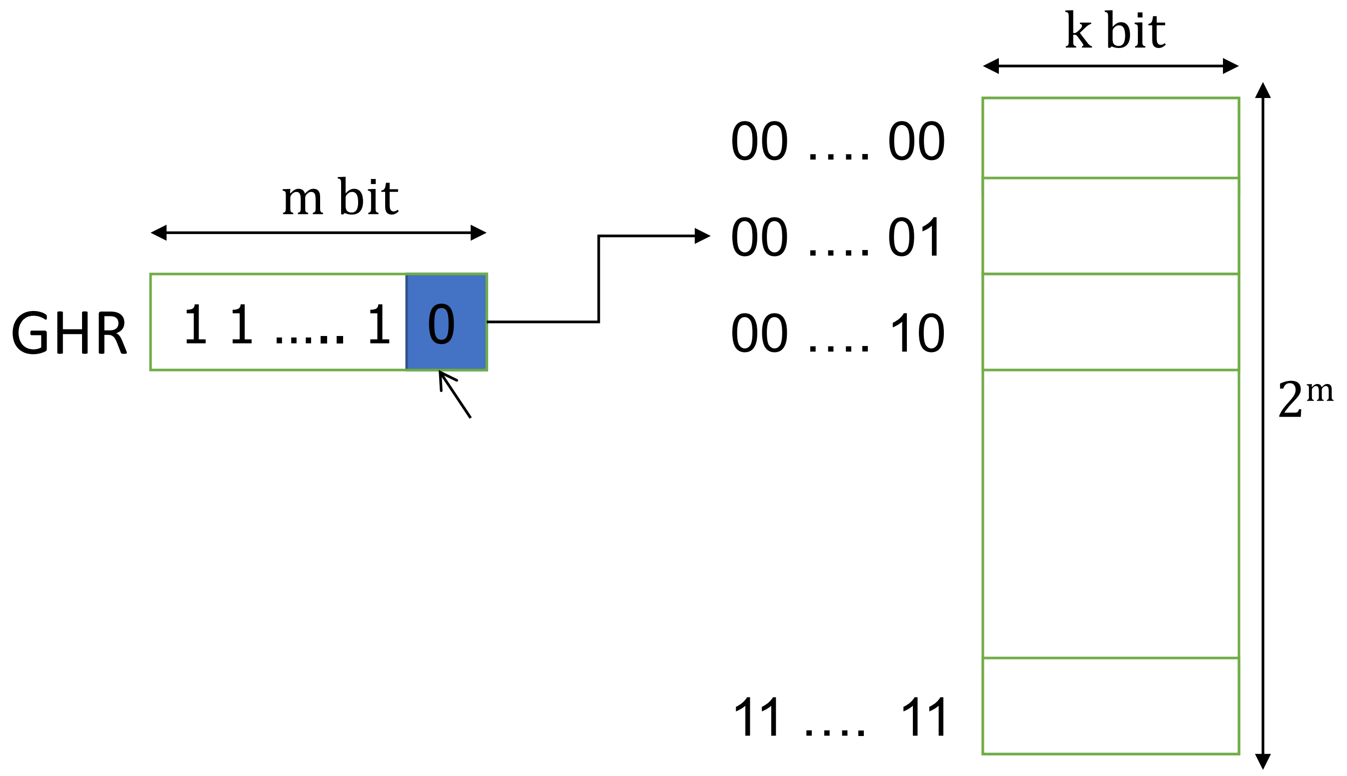
# Two Level Global Branch Prediction [MICRO '91]

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

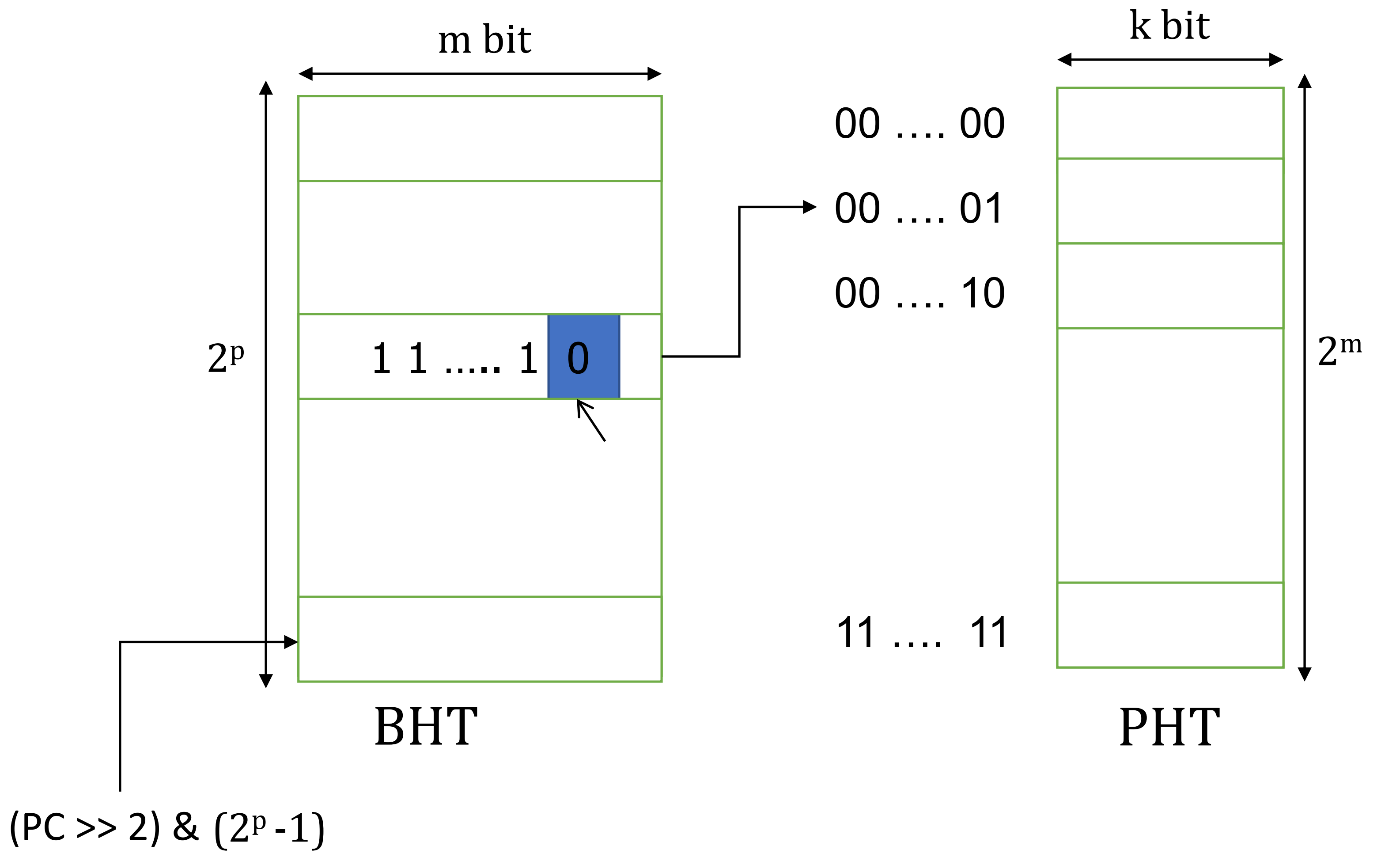


## PHT

- Table of saturating counters



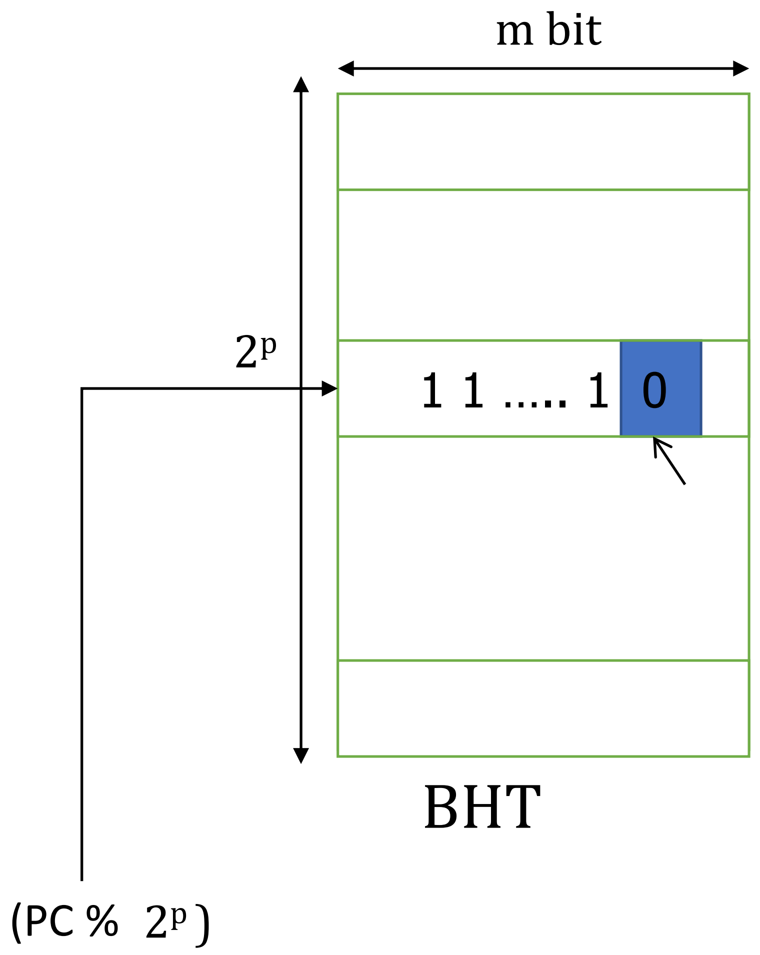
# GHR per Branch (Gain/Loss?)



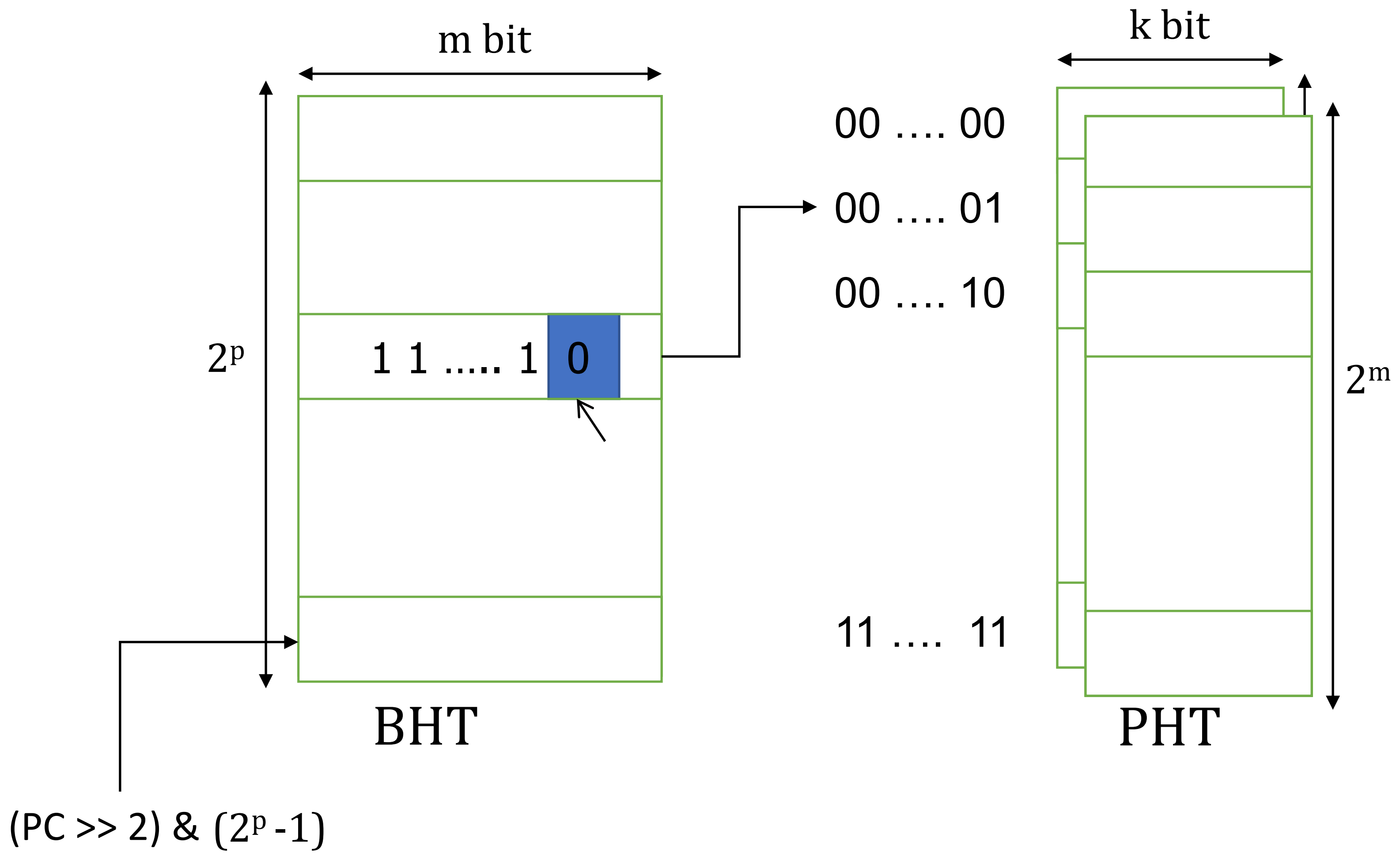
How large: k ?

Mostly K=2, m =12, how large m?

## Set of Branches – One Register



# What if One Branch -> One History -> One PHT ?



# Interference in Tables

- Sharing the PHTs between histories/branches leads to interference
    - Different branches map to the same PHT entry and modify it
    - Interference can be positive, negative, or neutral
- Interference can be eliminated by dedicating a PHT per branch

    -- Too much hardware cost
- How else can you eliminate

or reduce interference?

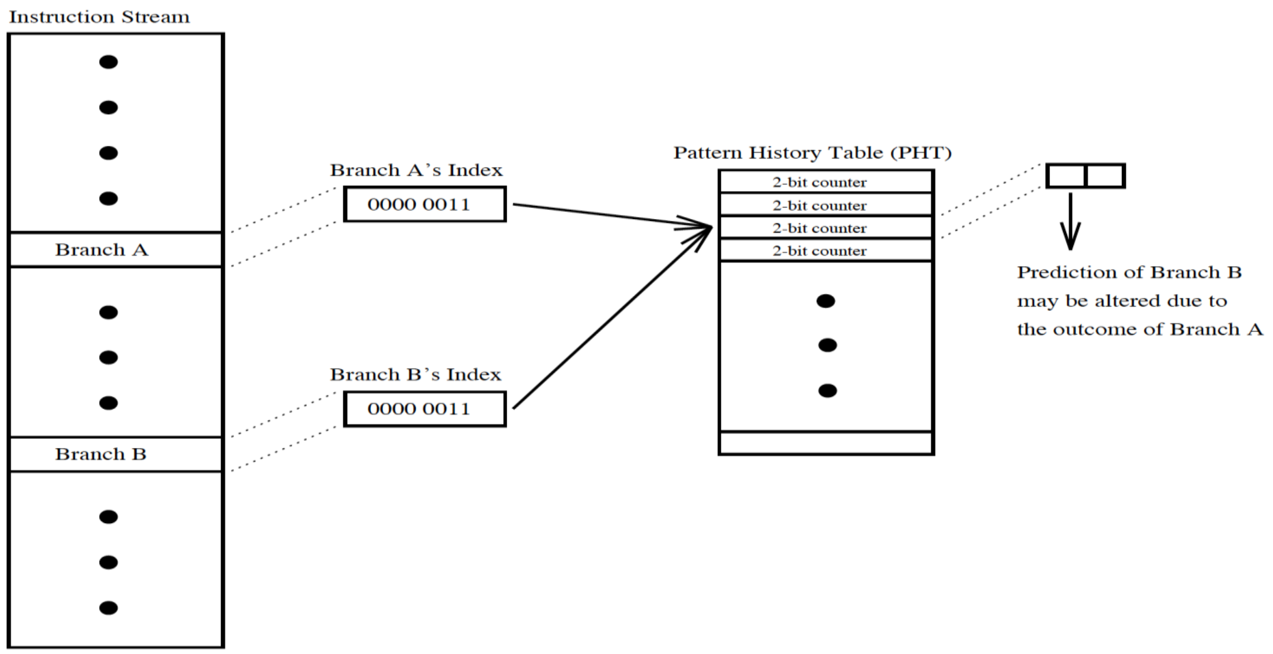
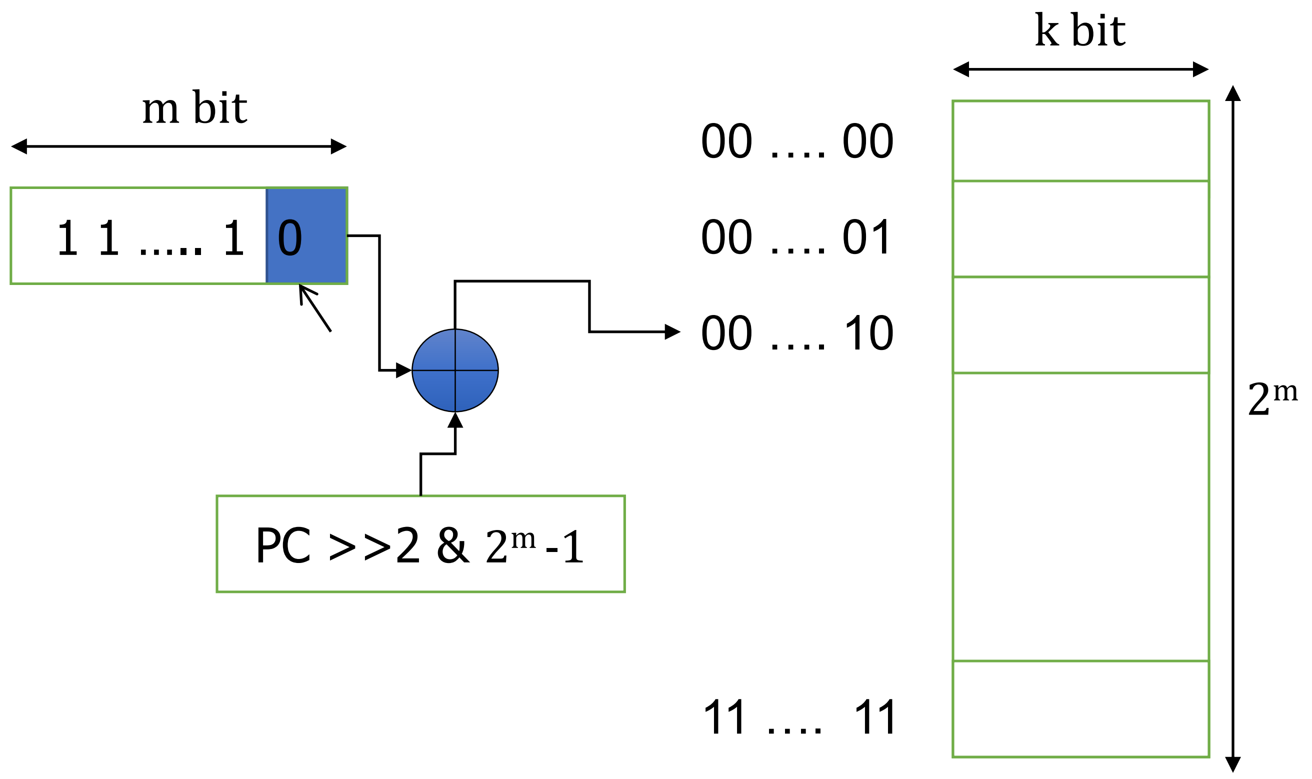


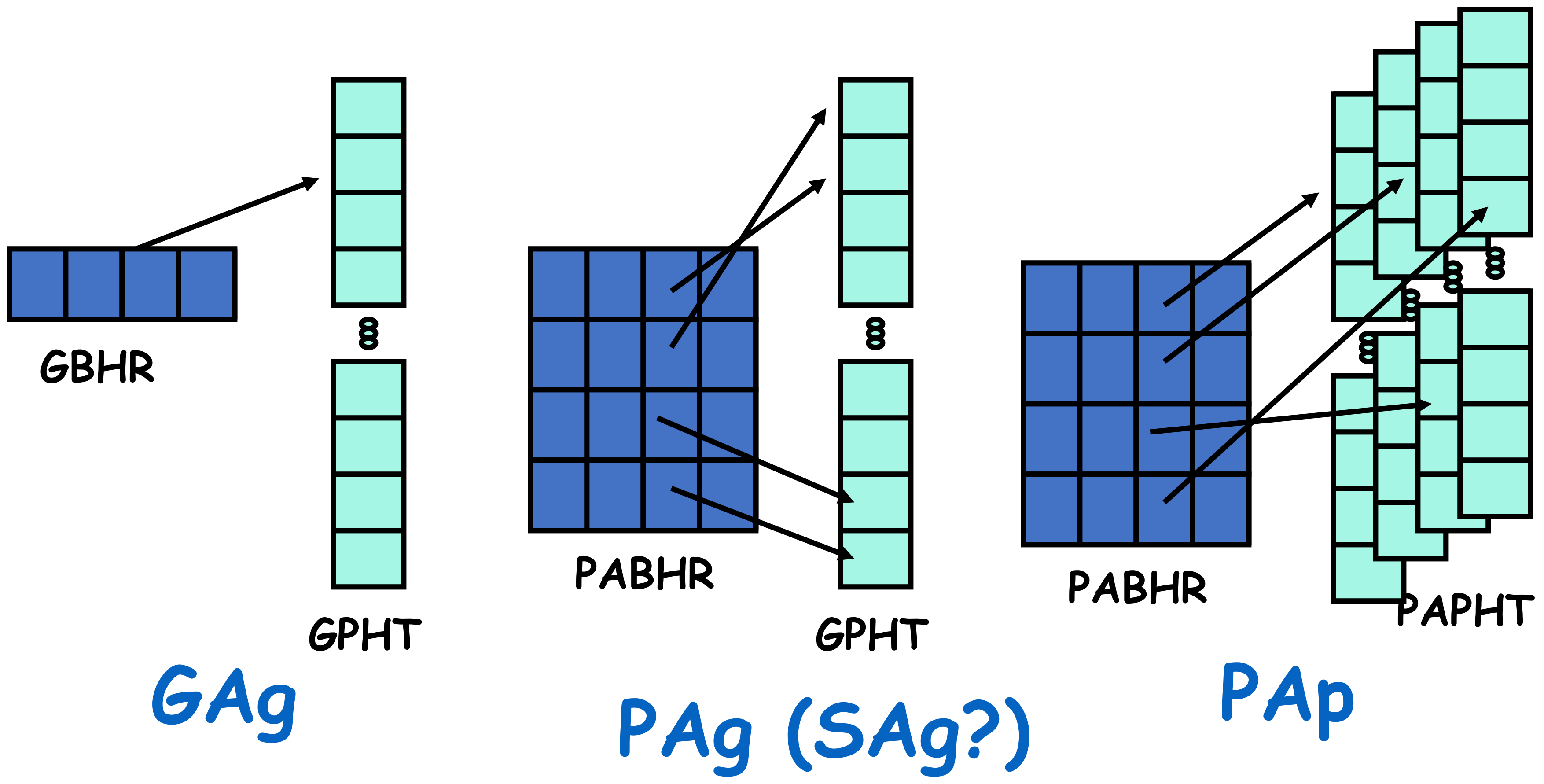
Figure 2: Interference in a two-level predictor.

# GShare



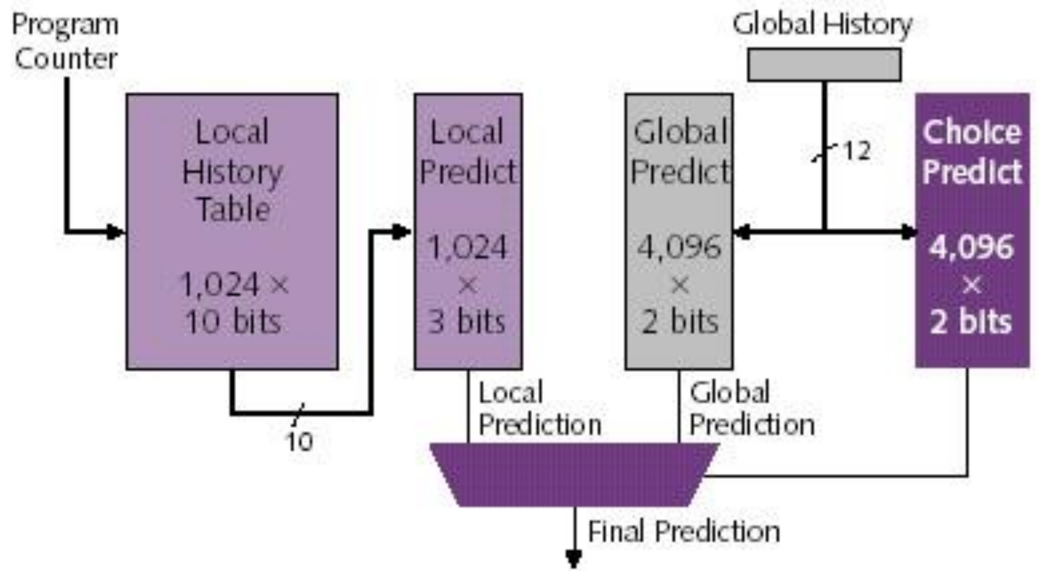
For a given history and for a given branch (PC) counters are trained

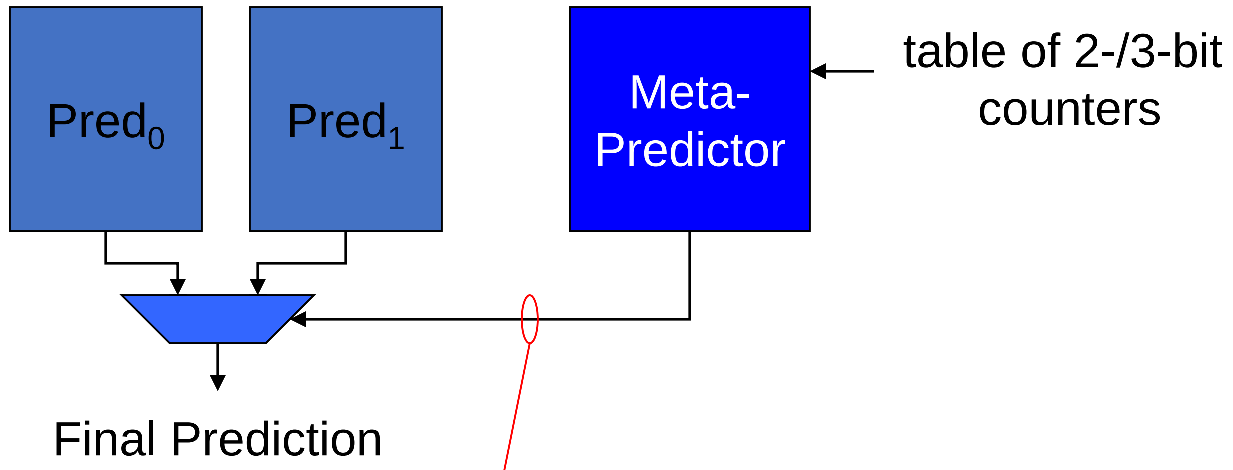
# Y & P Classification [MICRO 91]



- GAg: Global History Register, Global History Table
- PAg: Per-Address History Register, Global History Table
- PAp: Per-Address History Register, Per-Address History Table

# Tournament Predictor





If meta-counter MSB = 0, use $\text{pred}_0$ else use $\text{pred}_1$

| $\text{Pred}_0$ | $\text{Pred}_1$ | Meta Update |
|---|---|---|
| ✗ | ✗ | --- |
| ✗ | ✓ | Inc |
| ✓ | ✗ | Dec |
| ✓ | ✓ | --- |

## State-of-the-Art

### State of the art: Neural vs. TAGE

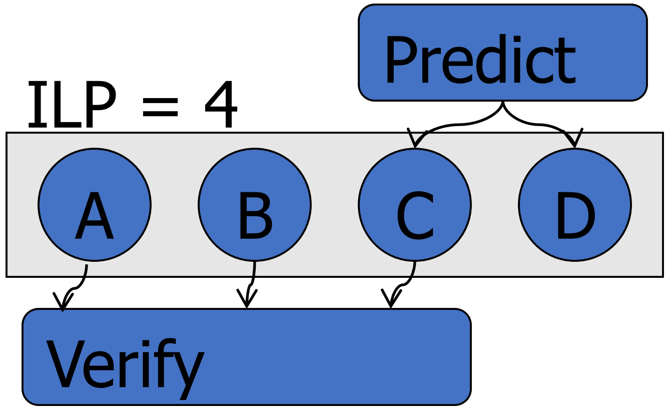
1970: Flynn
1972: Riseman/Foster
1979: Smith Predictor
1991: Two-level prediction
1993: gshare, tournament
1996: Confidence estimation
1996: Vary history length
1998: Cache exceptions
2001: Neural predictor
2004: PPM
2006: TAGE
2016: Still TAGE vs Neural

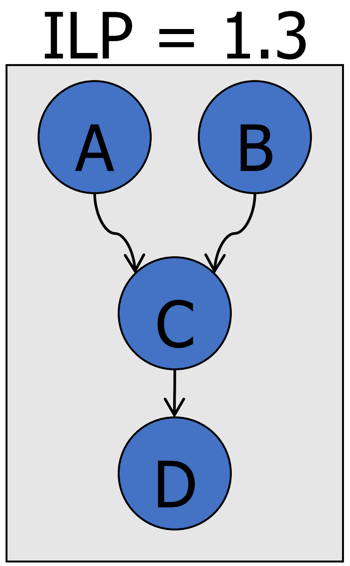
- Neural: AMD, Samsung
- TAGE: Intel?, ARM?
- Similarity
  - Many sources or “features”
- Key difference: how to combine them
  - TAGE: Override via partial match
  - Neural: integrate + threshold
- Every CBP is a cage match
  - Andre Seznec vs. Daniel Jimenez

## Spectre and Meltdown

https://meltdownattack.com/

# What about Values? And Why Values?





What is value prediction?

1. Generate a speculative value (predict)
2. Consume speculative value (execute)
3. Verify speculative value (compare/recover)

Goal: performance, i.e. expose more ILP

## Branch vs Value

- Predict result values of executing instruction
- Comparison to Branch Prediction
  - Branch Prediction
    - Single bit prediction (taken or not-taken)
    - Speed-up by solving control dependencies
    - Small penalty when prediction fails
  - Value Prediction
    - Multi-bit (32 or 64) prediction
    - Speed-up by solving data-dependencies
    - Large penalty when prediction fails

## Branch vs Value

- Most branch predictors are history-based
  - Simple predictors (e.g. last two values) have high hit-ratio.
  - SOTA predictor shows very high hit-ratio.

- Value prediction is much difficult
  - Locality of result values exists but far less than that of branch
  - About 50% executed instructions output last values and 80% instructions output one of 16 latest results [Lipasti+,96]
  - Much lower hit-ratio than branch predictors

## Branch vs Value

- Miss penalty

Branch prediction << Value prediction

  - Latency of normal instruction < branch instruction

  - Number of canceled instruction is much larger than that in branch prediction

## ACKS

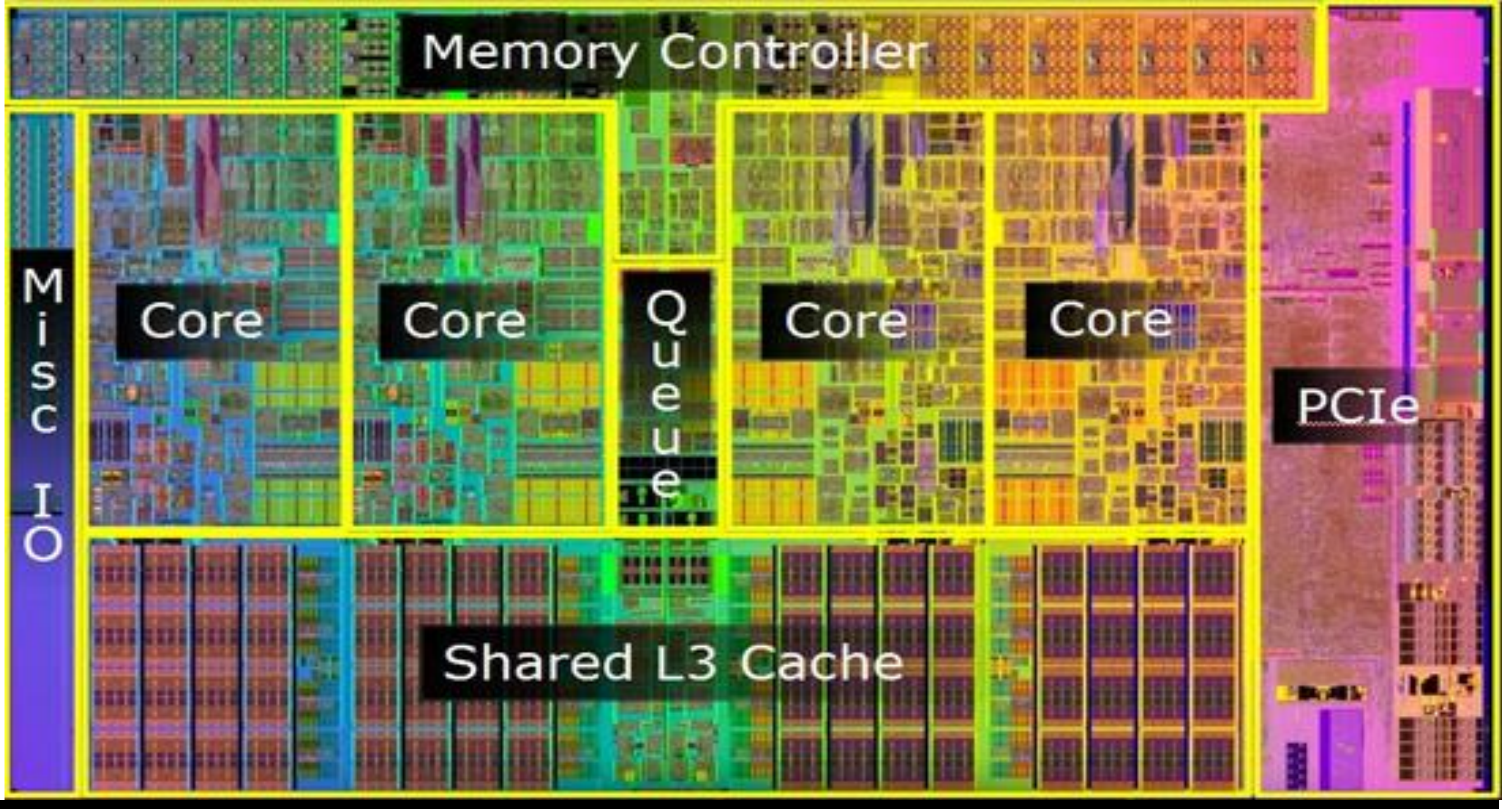
- Some of the slides are adapted and modified from Joel Emer, Arvind, Yale Patt, John Kubiatowicz, Onur Mutlu, Krste Asanovic, Mattan Erez, Rajeev Balasubramonian, and Mainak Chaudhuri, Mikko Lipasti, and Kei Hiraki

# CASH @CASS '18:

Biswa@CSE-IITK

image courtesy: India Today

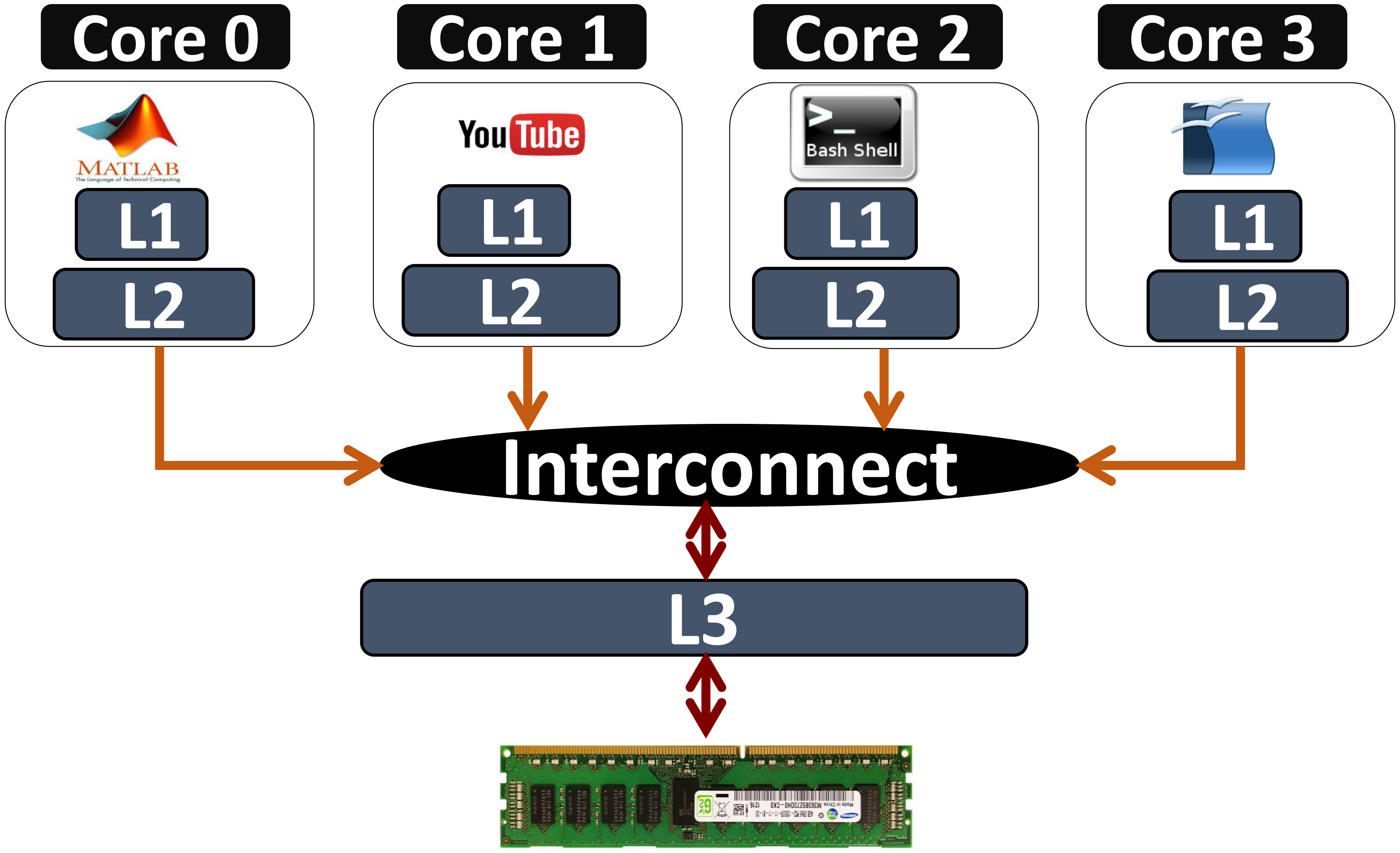
# CACHES @CASS '18:



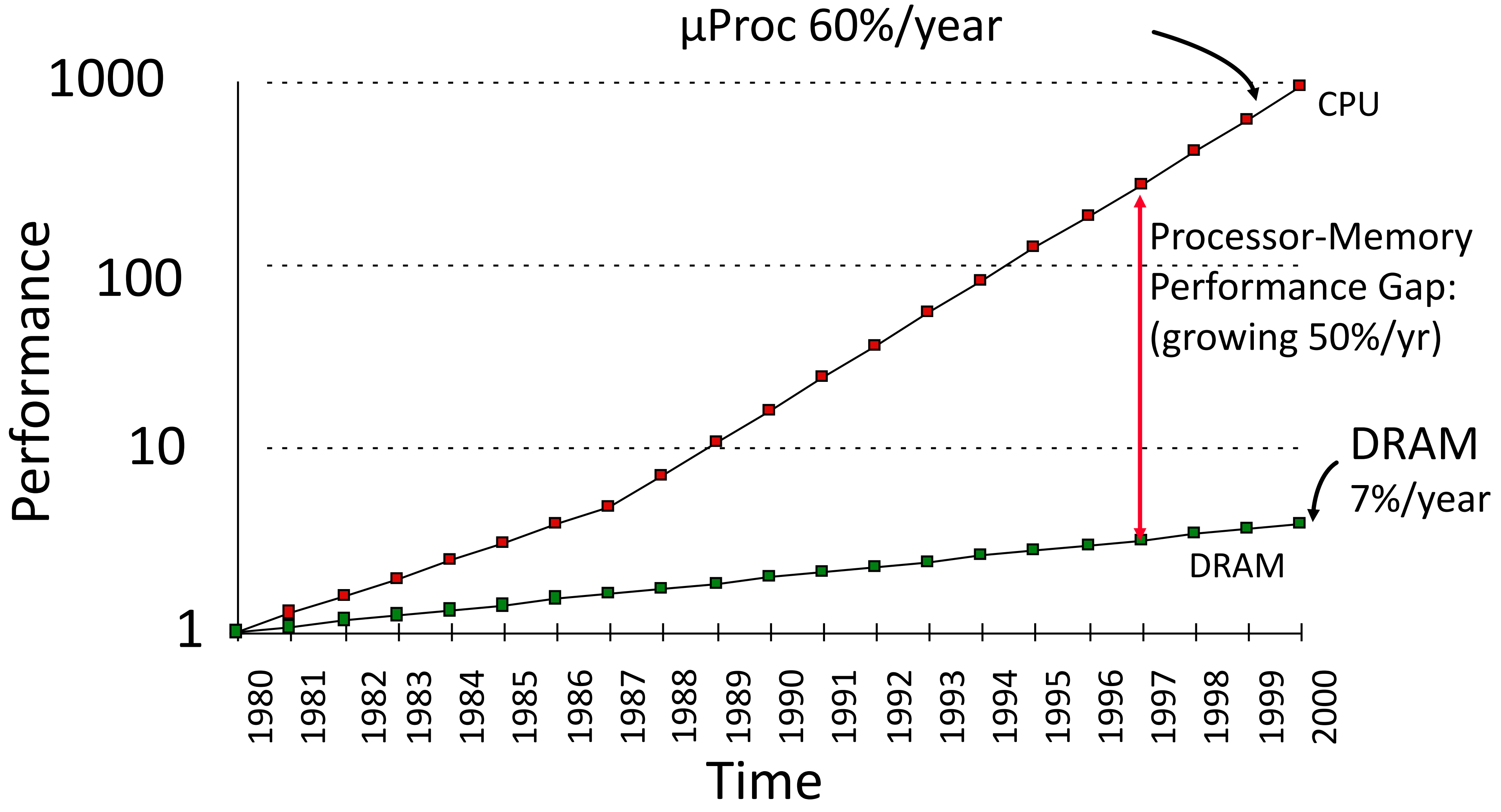
Biswa@CSE-IITK

image courtesy: Intel

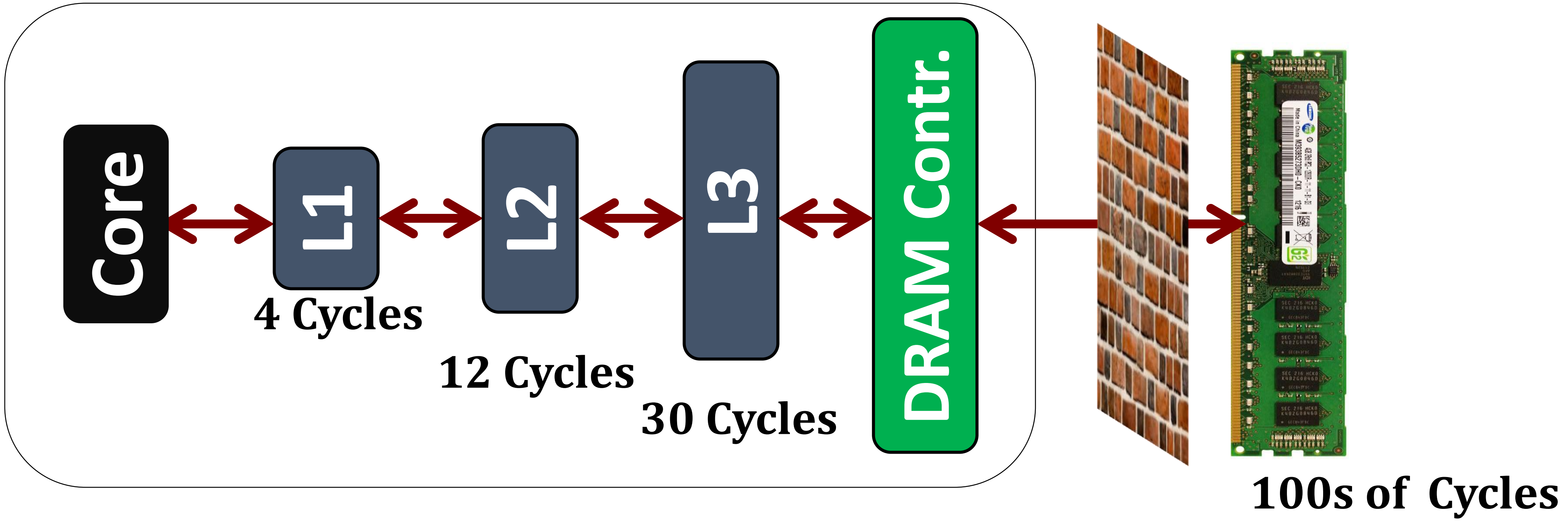
# CACHES: WHERE ARE THEY



# Memory Wall Problem



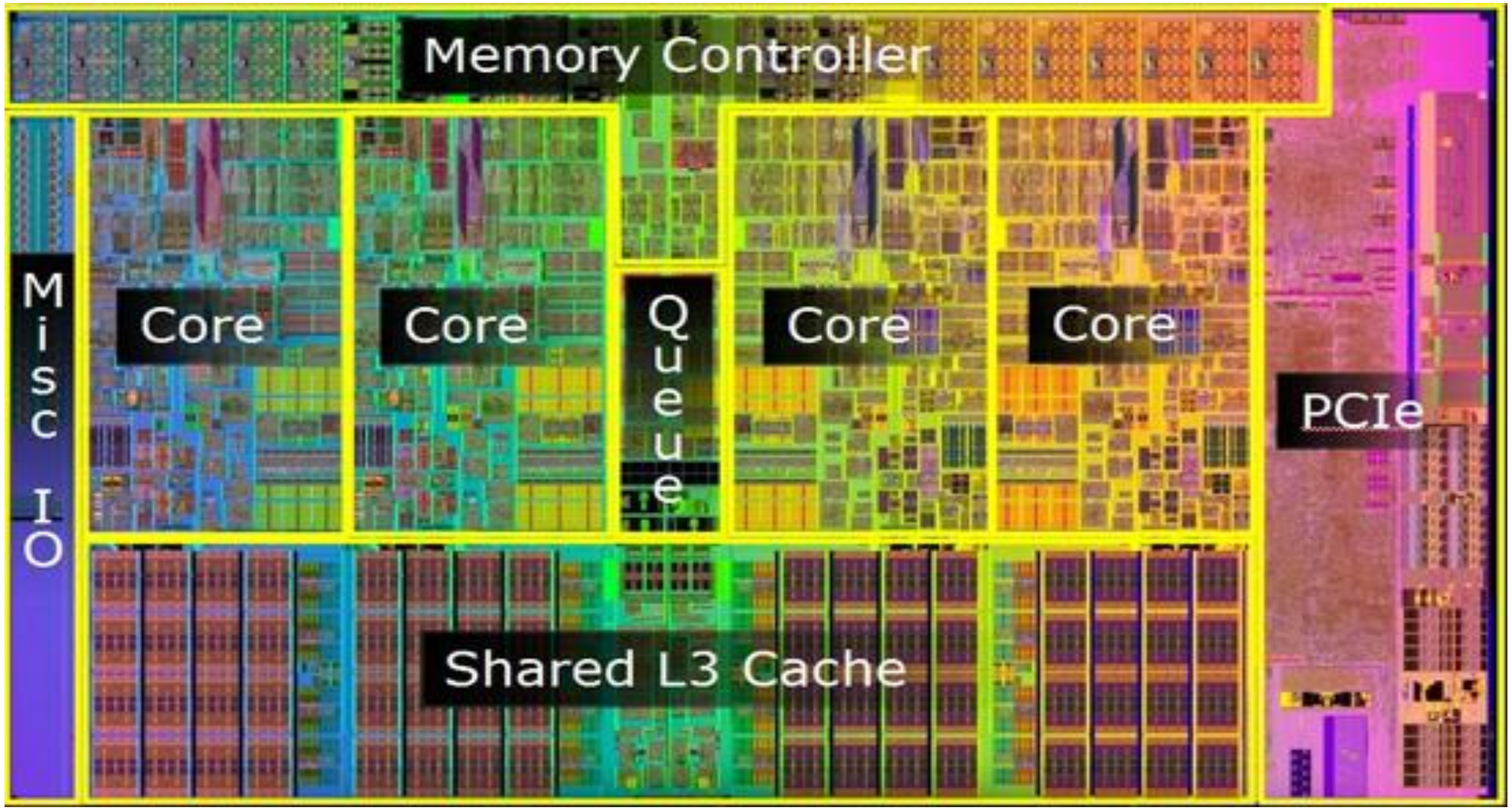
# WHY ARE THEY?



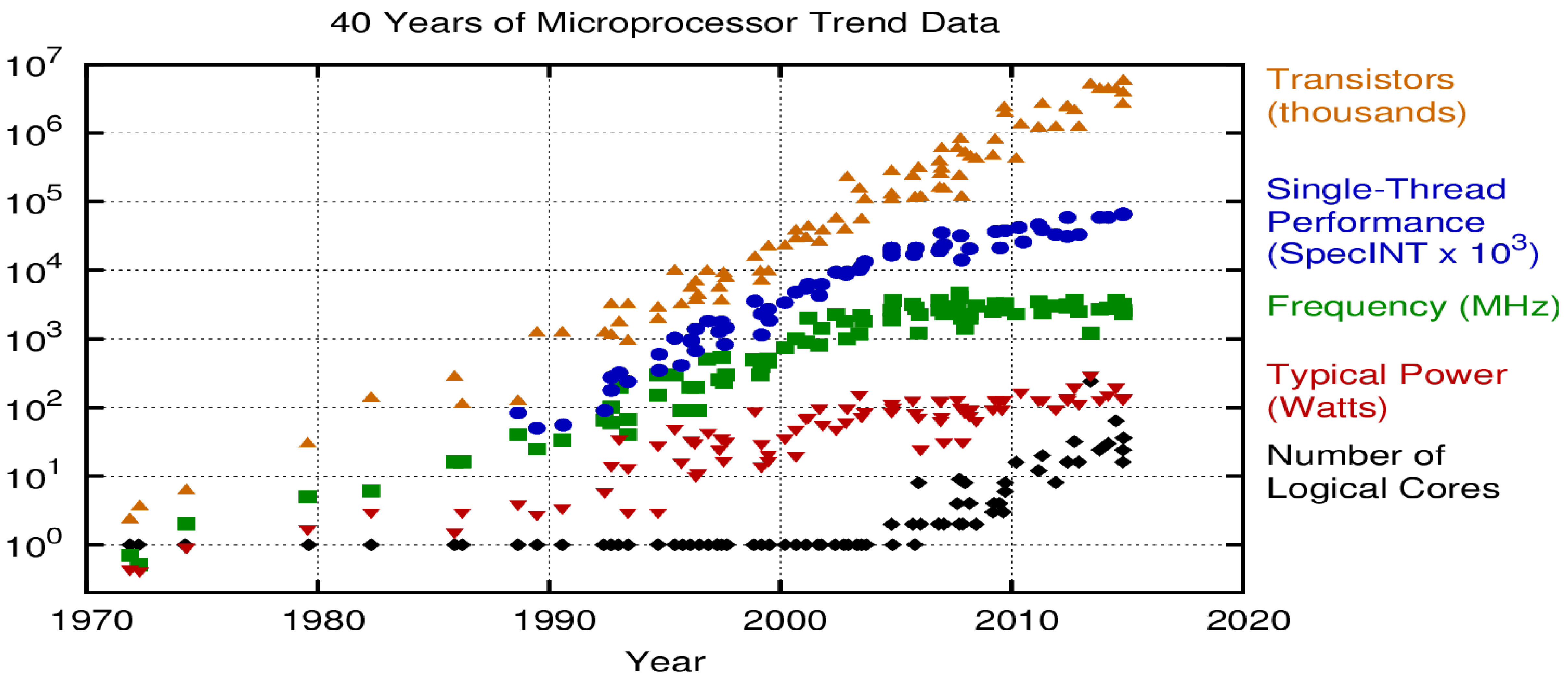
**Latency wall**

**Bandwidth wall**

## SILICON VIEW

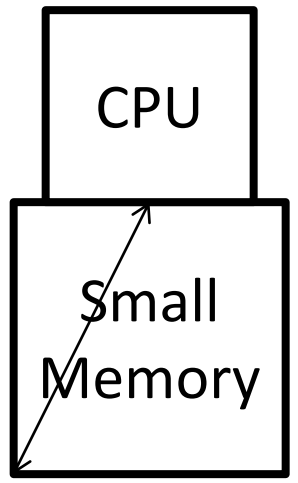


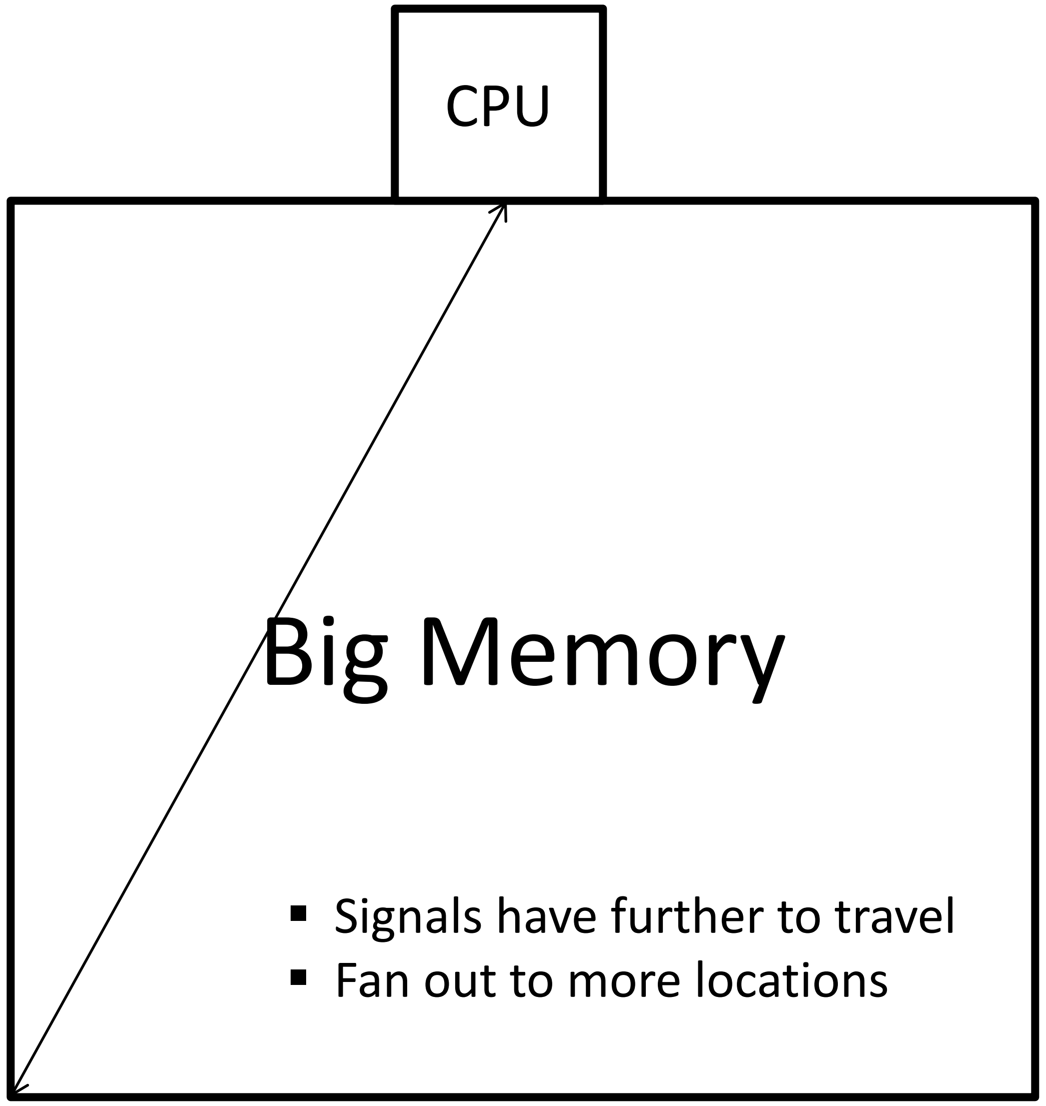
# CORE COUNT



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

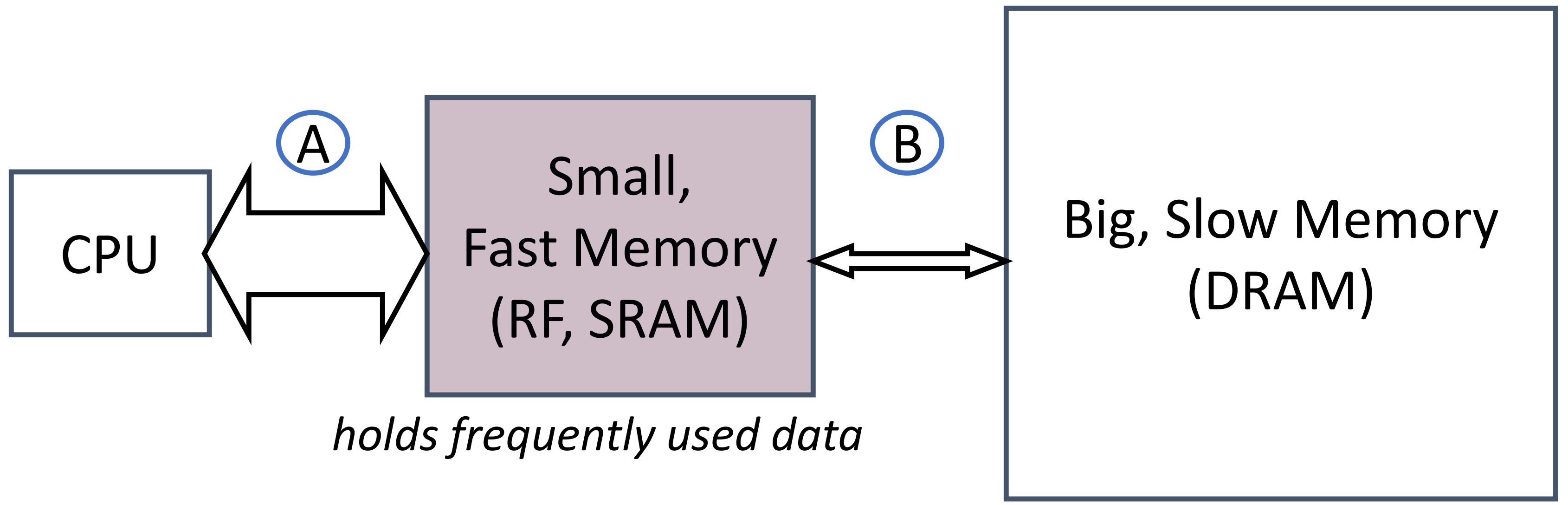
## BASICS FIRST: Size Affects Latency





- Signals have further to travel
- Fan out to more locations

# Memory Hierarchy



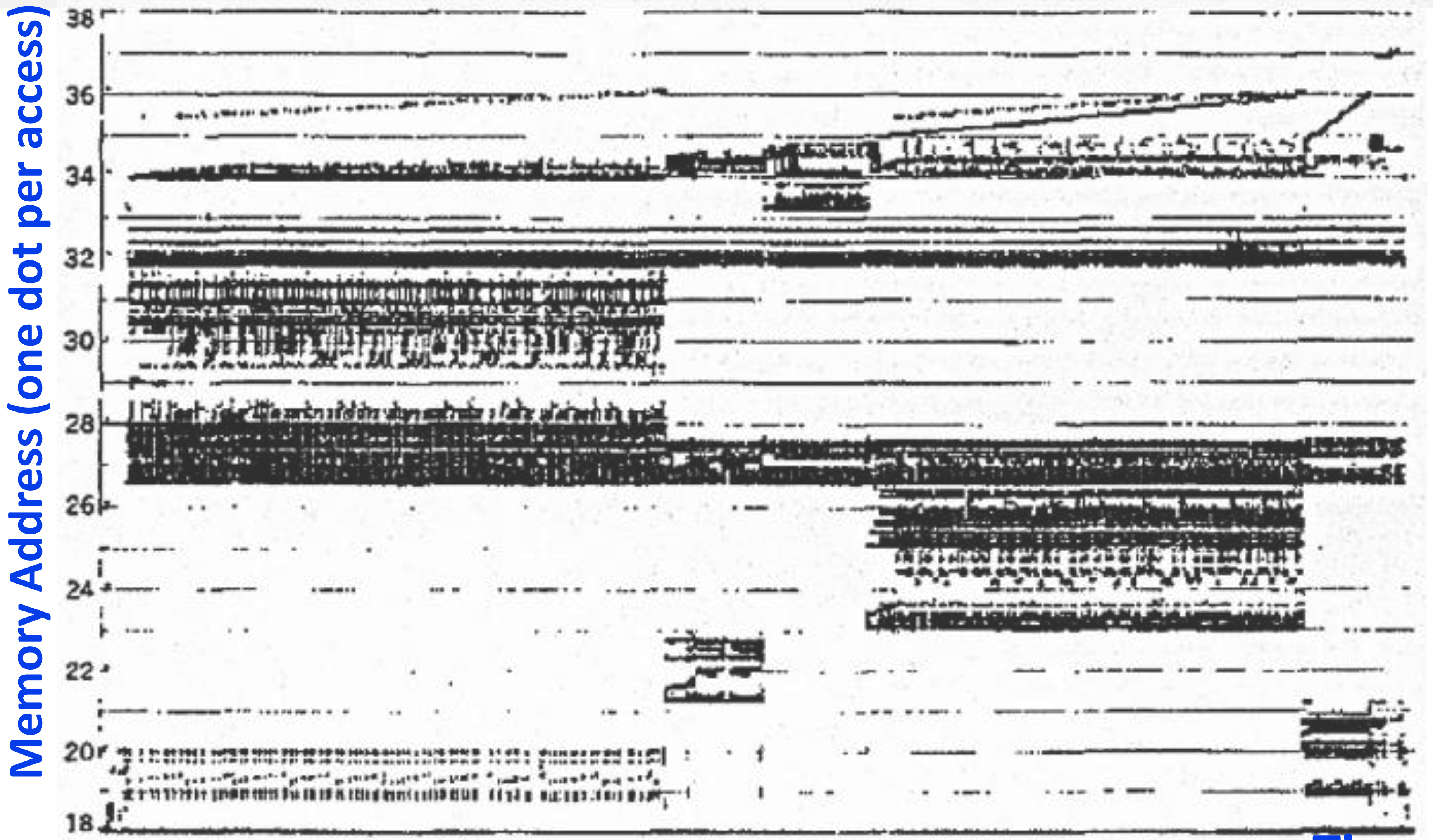
- *capacity*: Register << SRAM << DRAM
- *latency*: Register << SRAM << DRAM
- *bandwidth:* on-chip >> off-chip

On a data access:

*if* data $\in$ fast memory $\Rightarrow$ low latency access *(SRAM)*

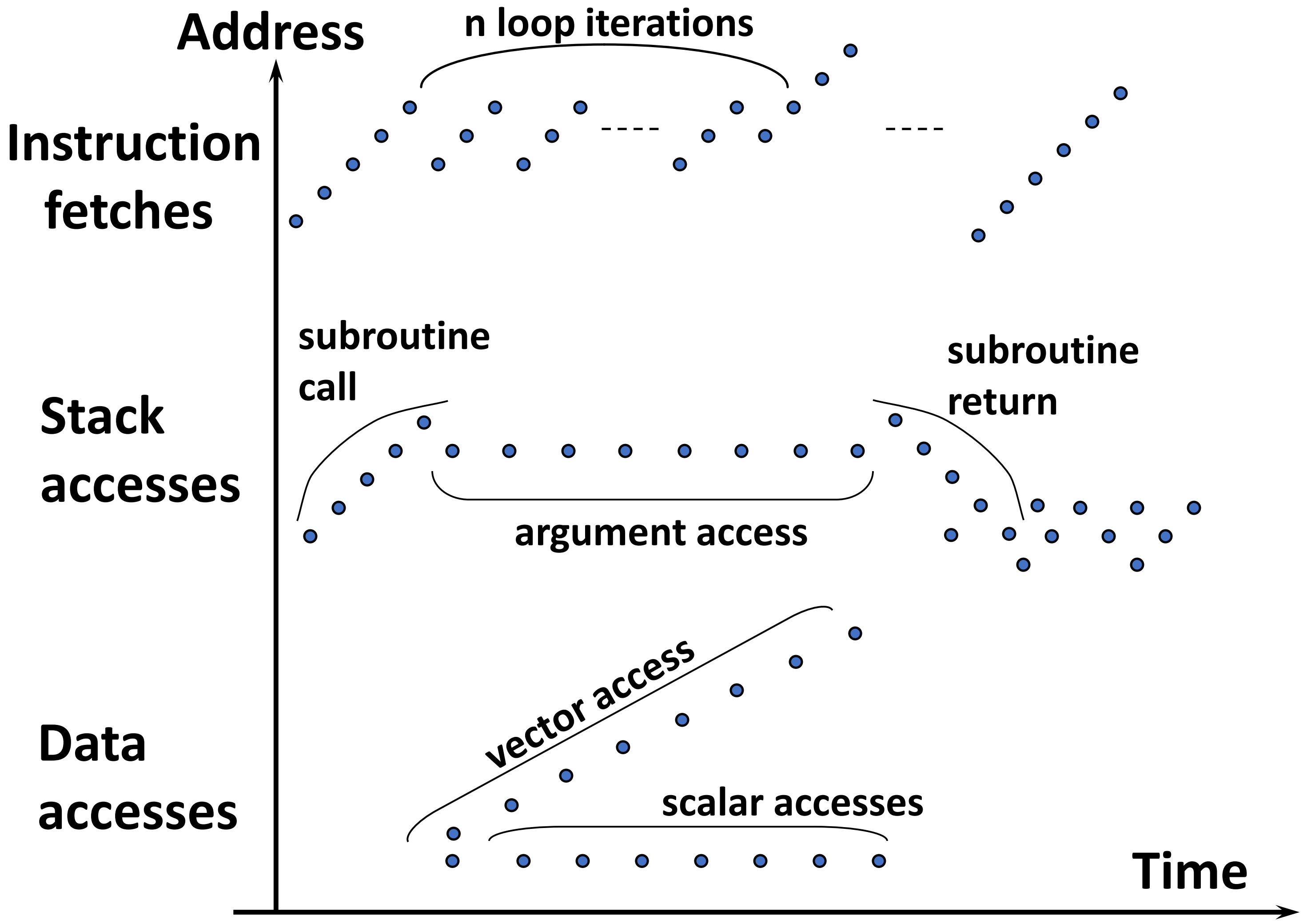
*if* data $\notin$ fast memory $\Rightarrow$ high latency access *(DRAM)*

# Access Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

# Examples

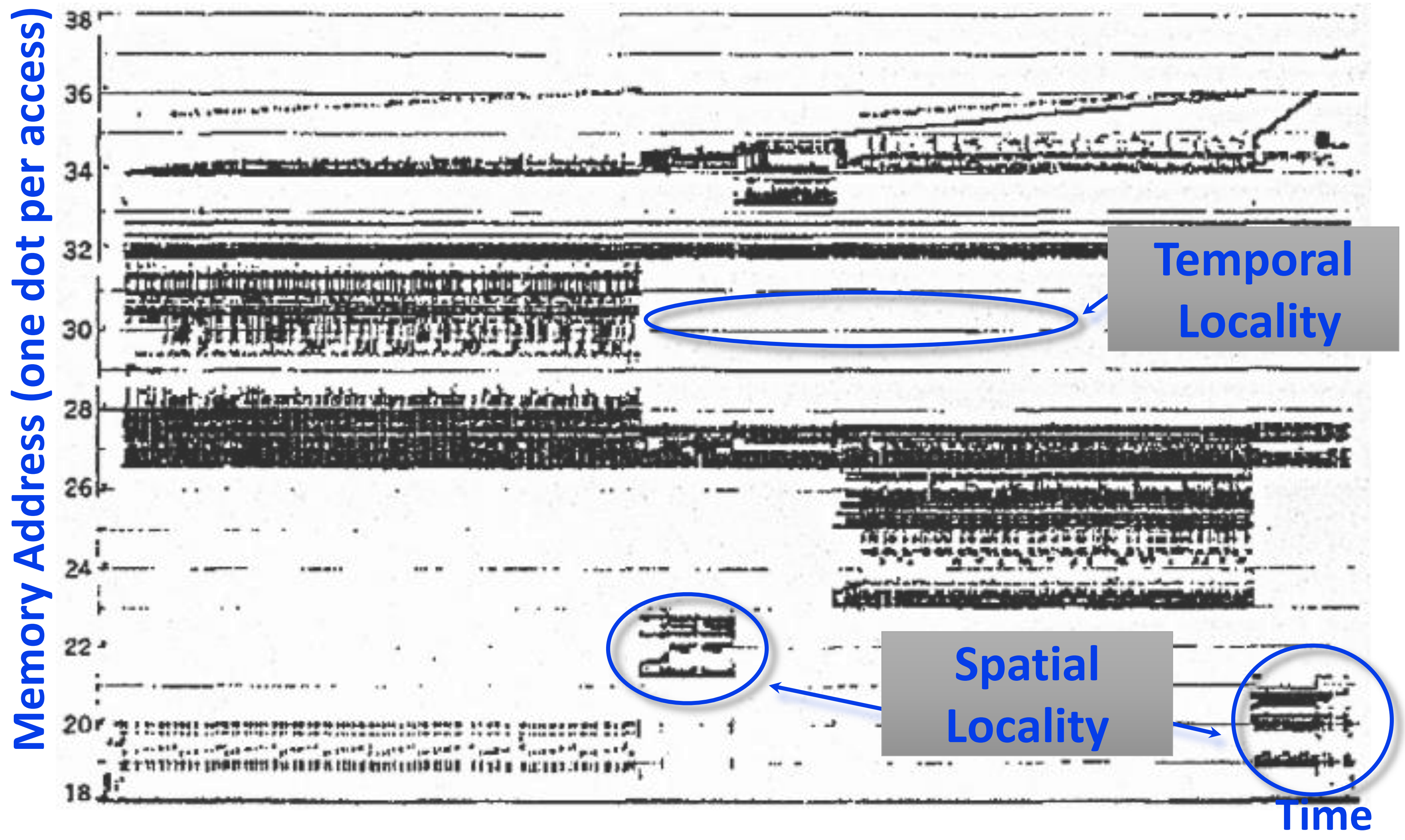

Address
n loop iterations
Instruction fetches
subroutine call
subroutine return
Stack accesses
argument access
vector access
scalar accesses
Data accesses
Time

## Locality of Reference

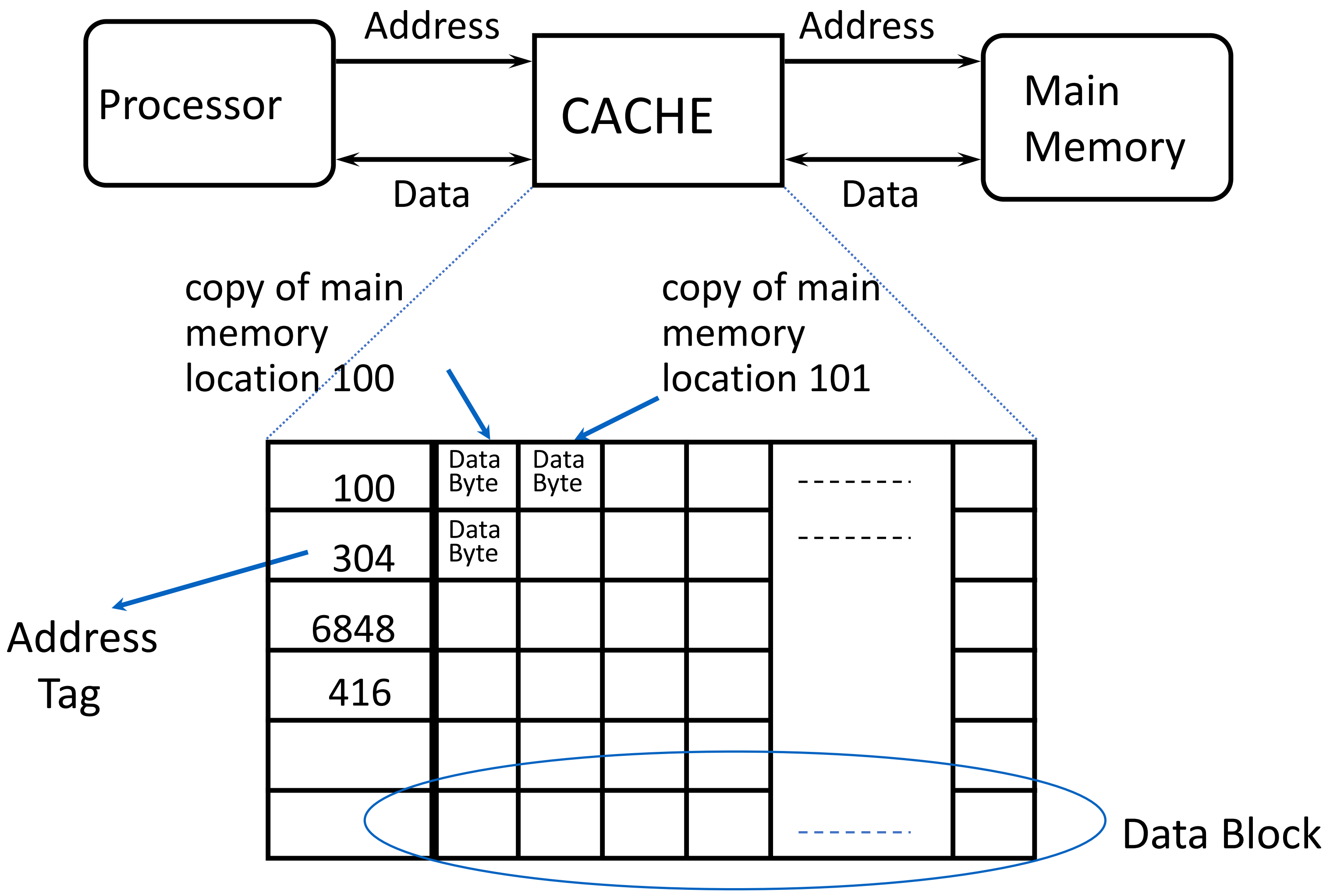
- **Temporal Locality**: If a location is referenced it is likely to be referenced again in the near future.

- **Spatial Locality**: If a location is referenced it is likely that locations near it will be referenced in the near future.

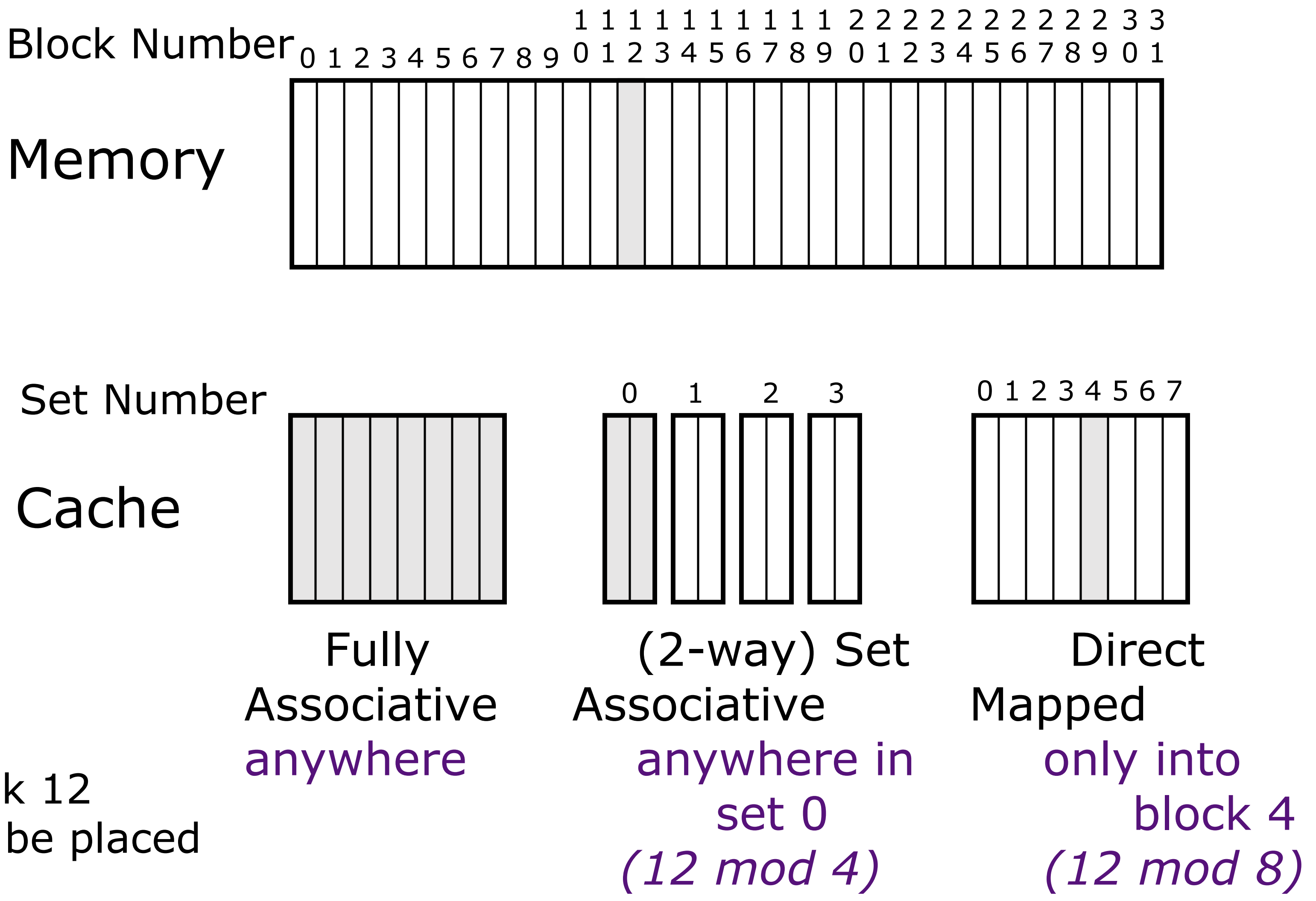
# Again


Memory Address (one dot per access)
38
36
34
32
30
28
26
24
22
20
18
Temporal Locality
Spatial Locality
Time

# Inside a Cache

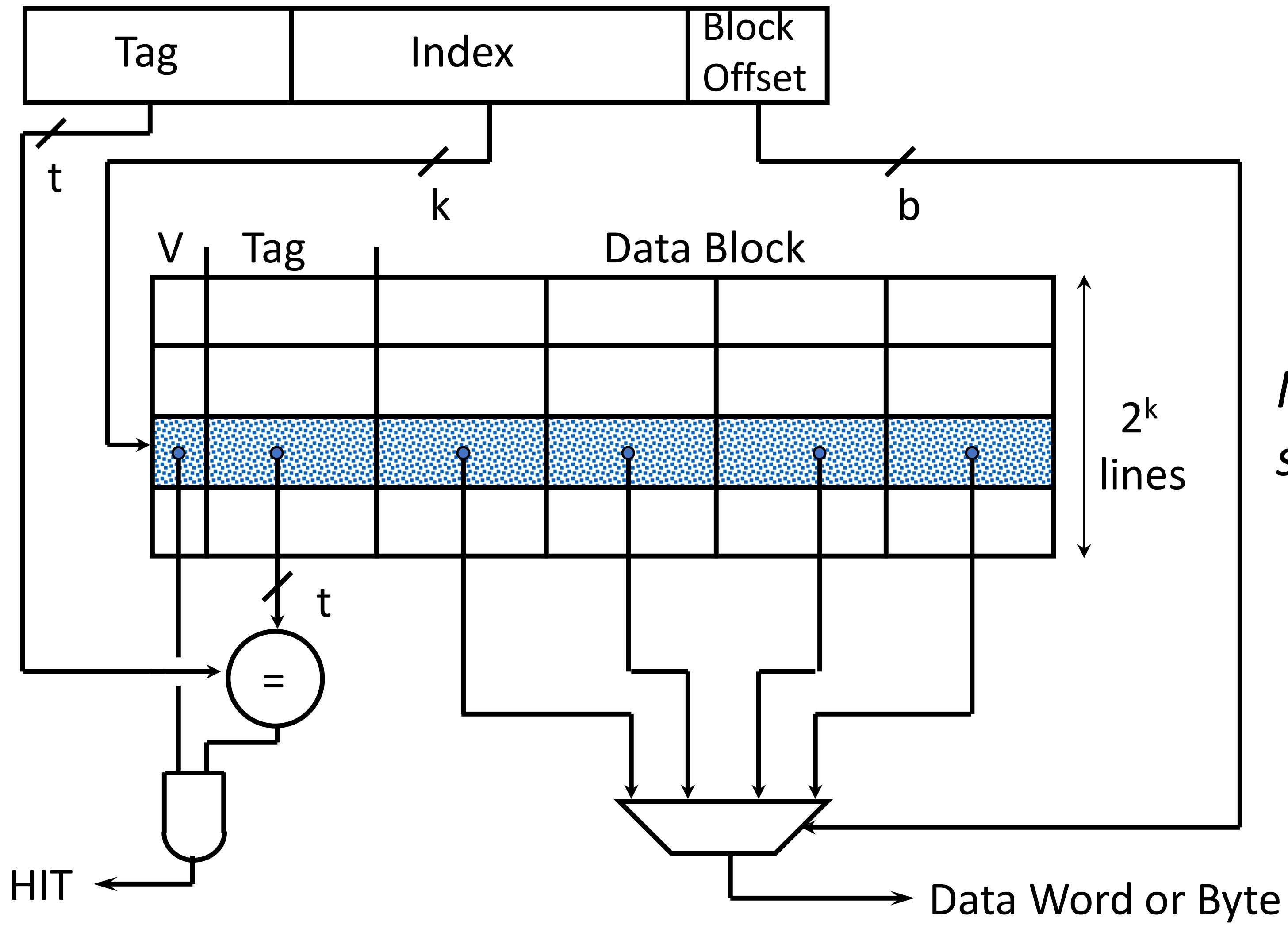


## Placement Policy



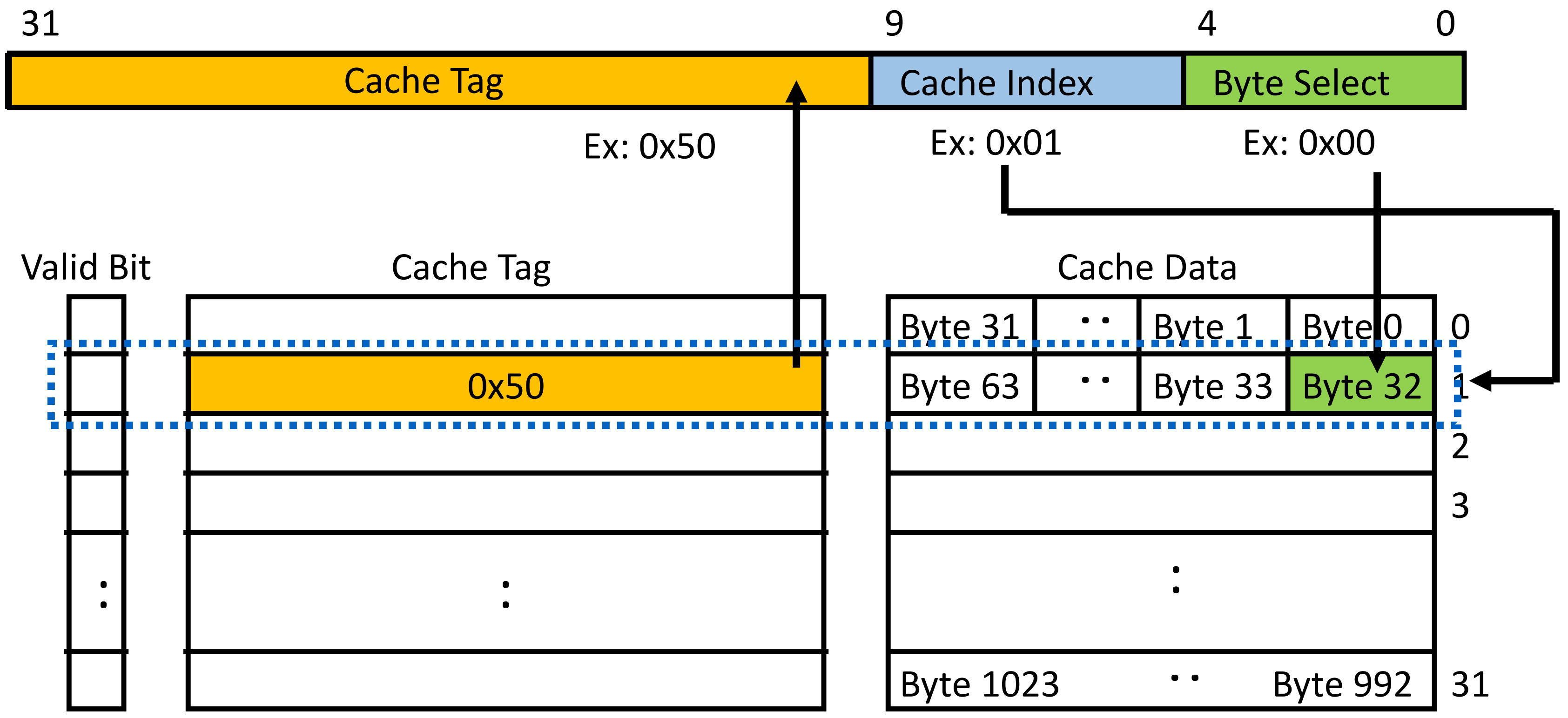
block 12
can be placed

# Direct Mapped



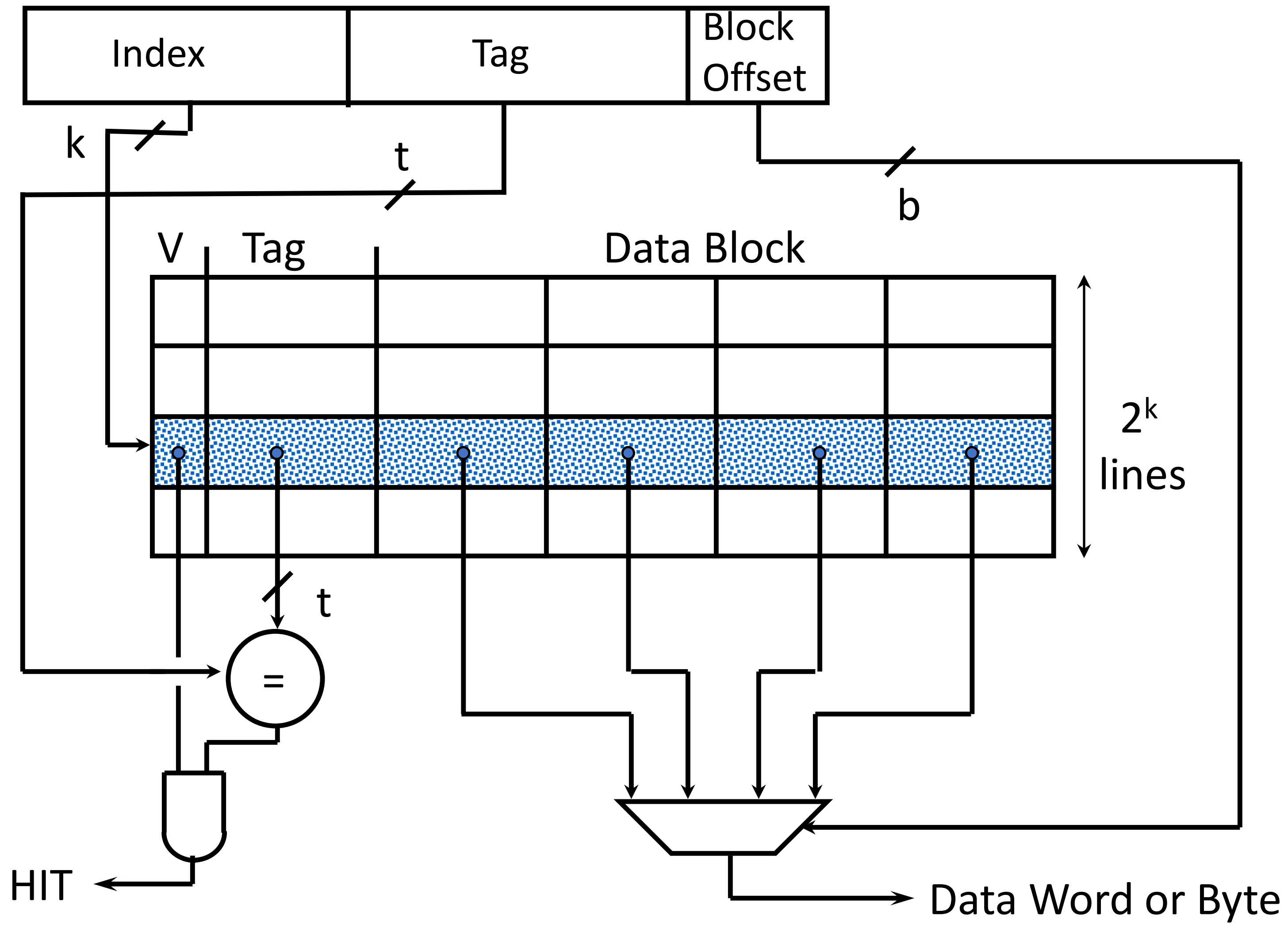
*In reality, tag-store is placed separately*

# An Example

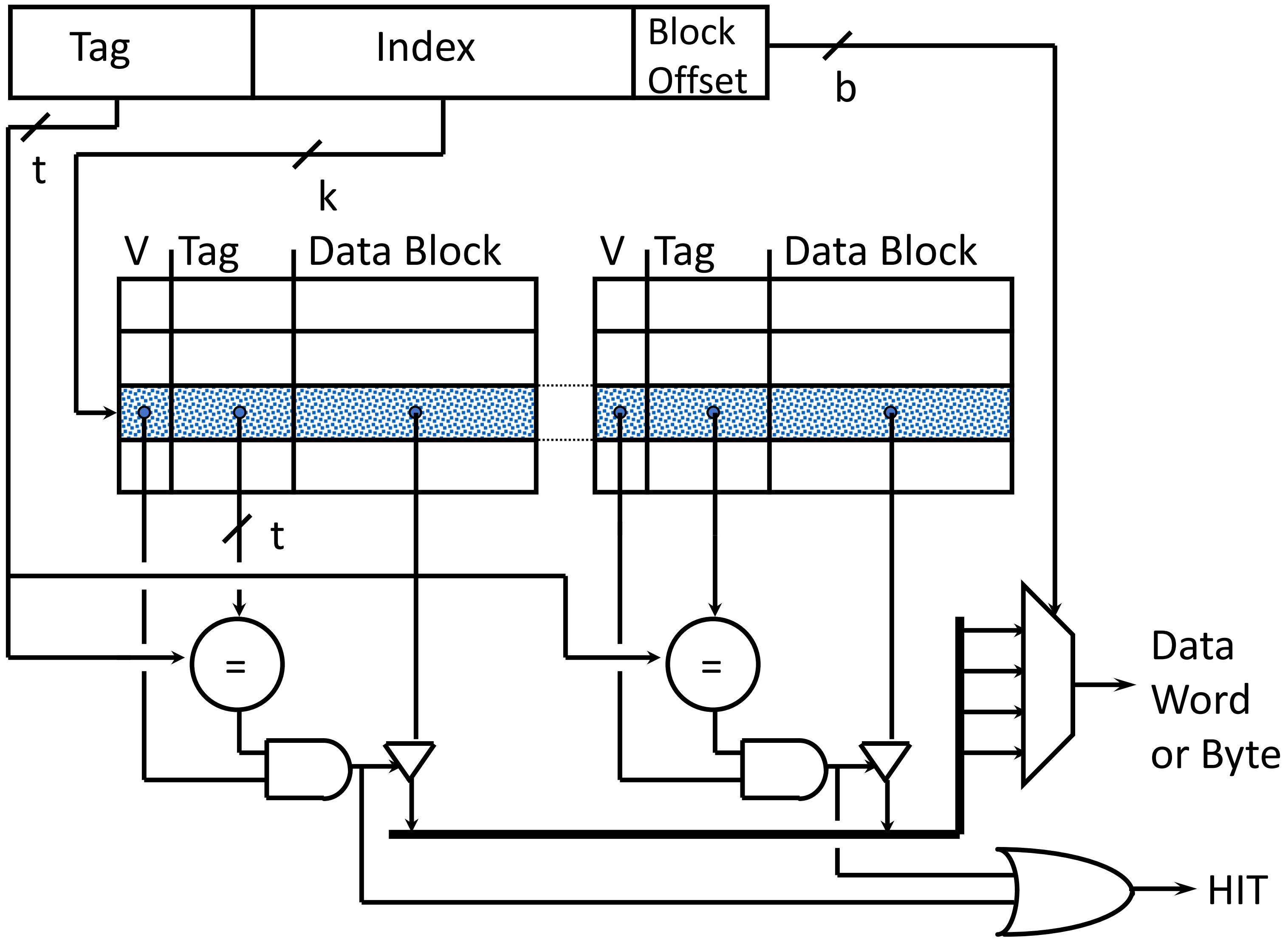


31
9
4
0
Cache Tag
Cache Index
Byte Select
Ex: 0x50
Ex: 0x01
Ex: 0x00
Valid Bit
Cache Tag
Cache Data
0x50
Byte 31
Byte 1
Byte 0
Byte 63
Byte 33
Byte 32
Byte 1023
Byte 992
0
1
2
3
31

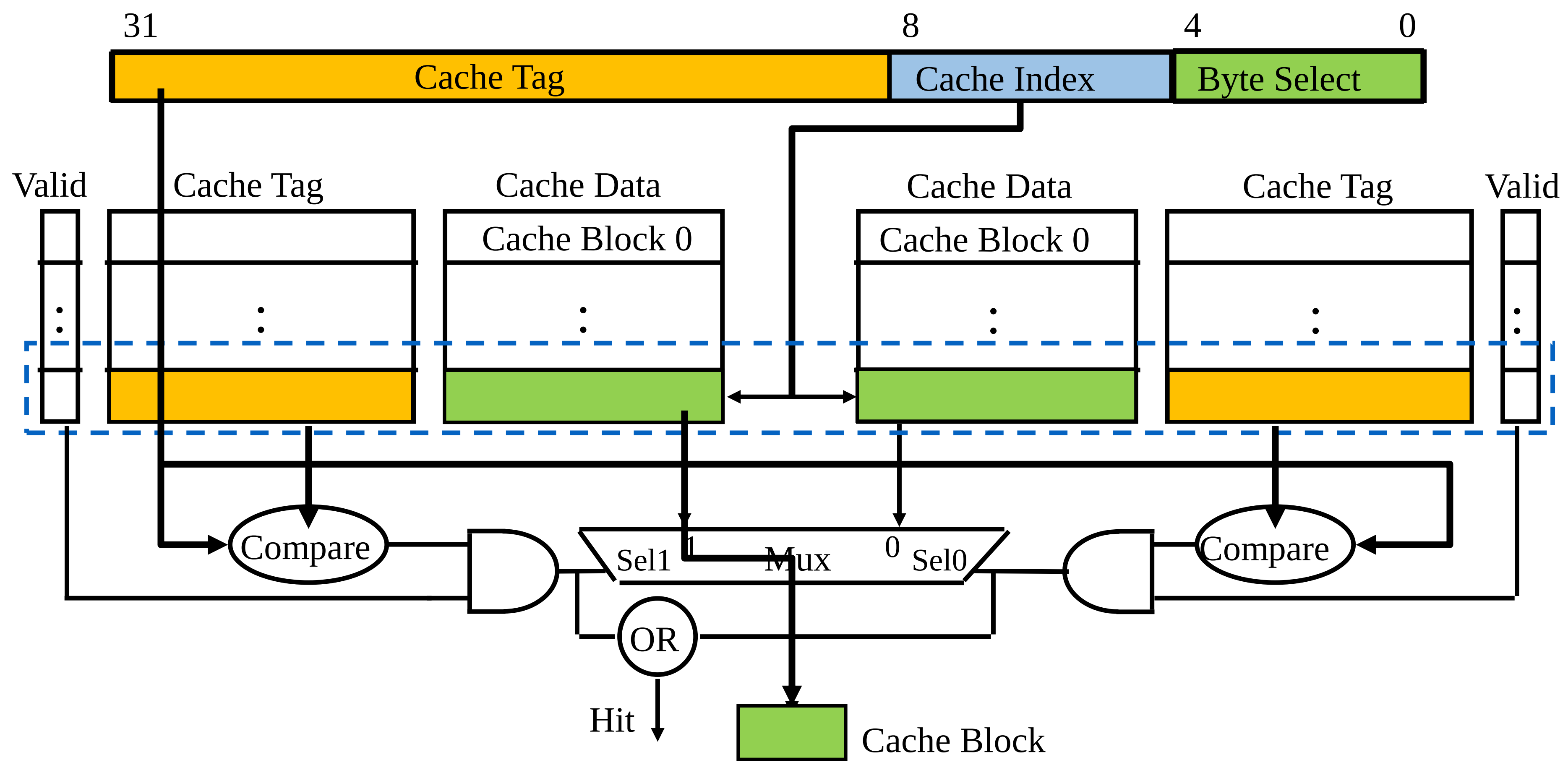
# High bits or Low bits



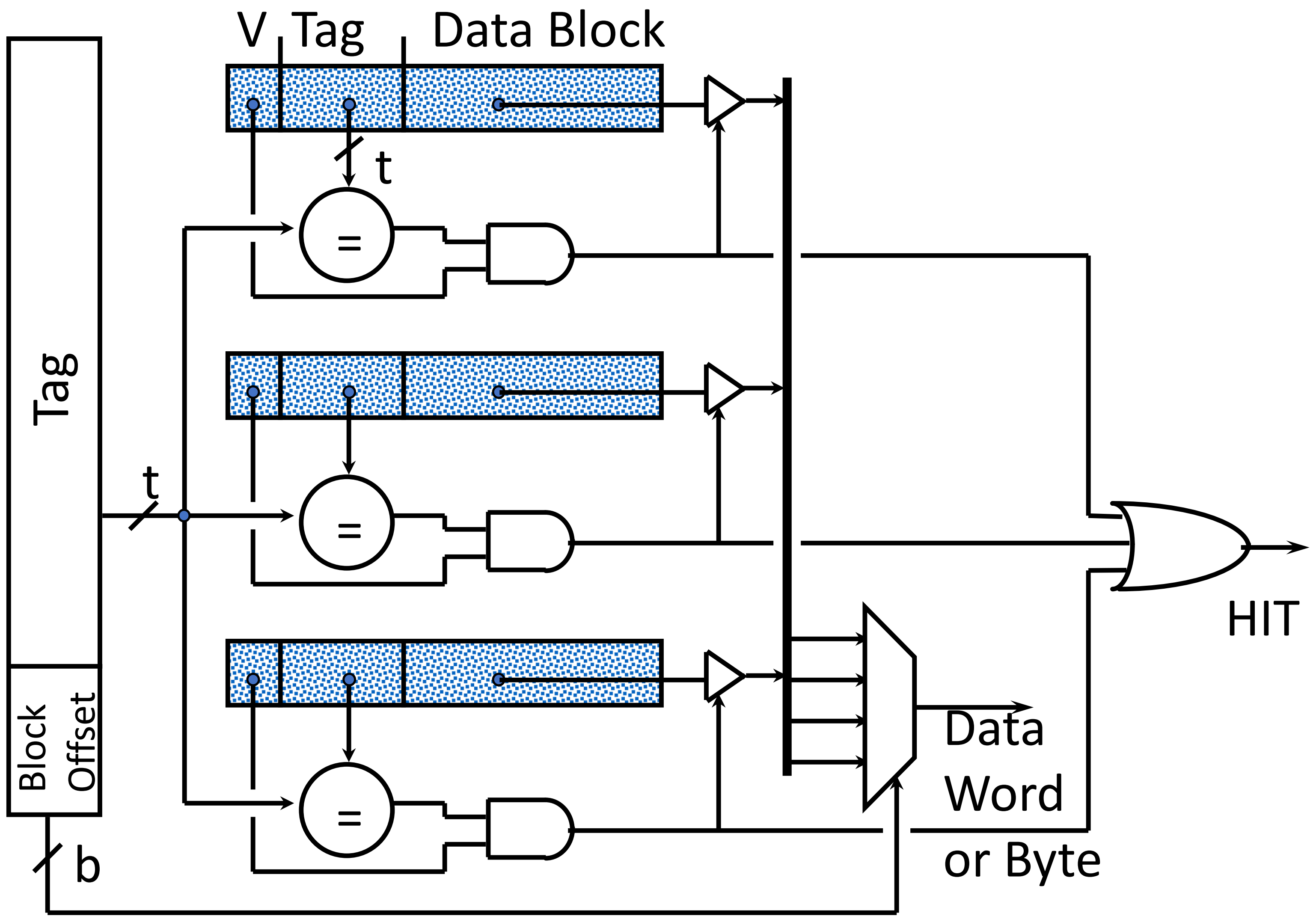
# Set-Associative



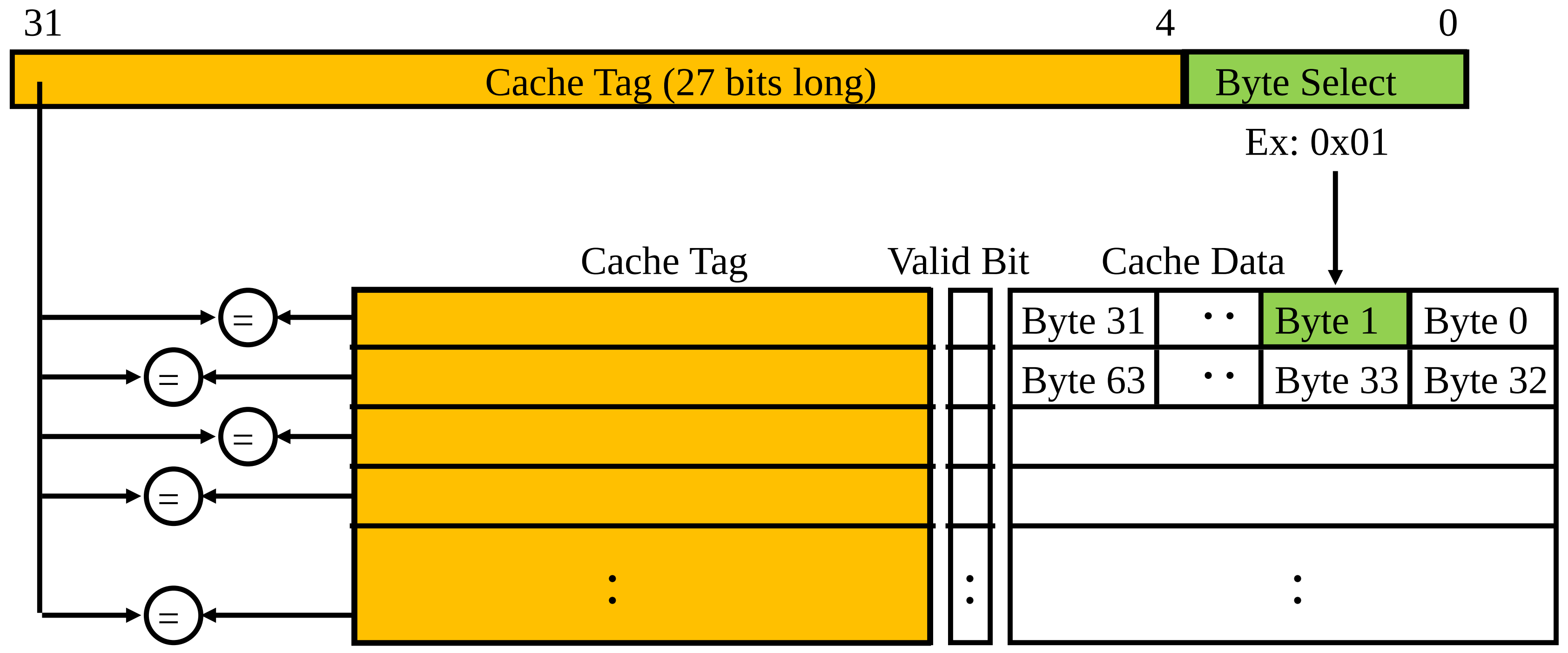
# An Example


31
8
4
0
Cache Tag
Cache Index
Byte Select
Valid
Cache Tag
Cache Data
Cache Block 0
Cache Data
Cache Block 0
Cache Tag
Valid
Compare
Sel1
1
Mux
0
Sel0
Compare
OR
Hit
Cache Block

## Fully-associative



# An Example

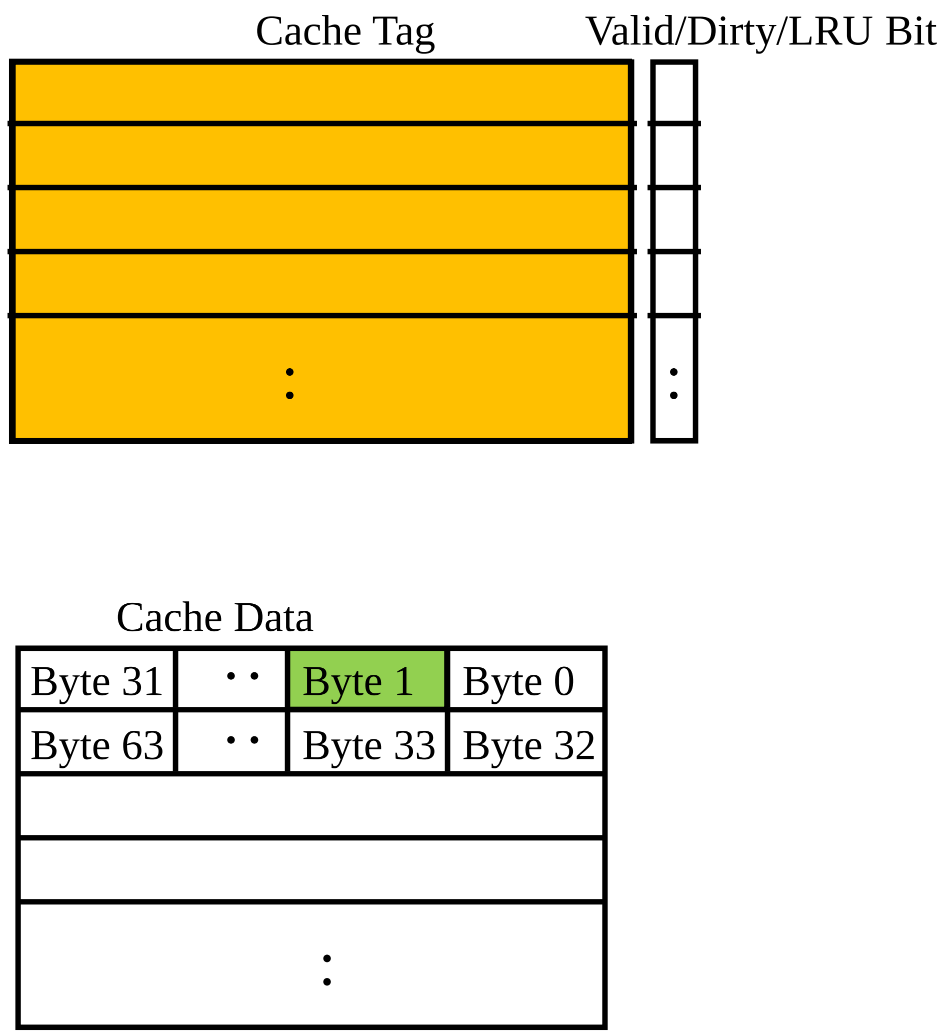


31
4
0
Cache Tag (27 bits long)
Byte Select
Ex: 0x01
Cache Tag
Valid Bit
Cache Data
Byte 31
Byte 1
Byte 0
Byte 63
Byte 33
Byte 32
=

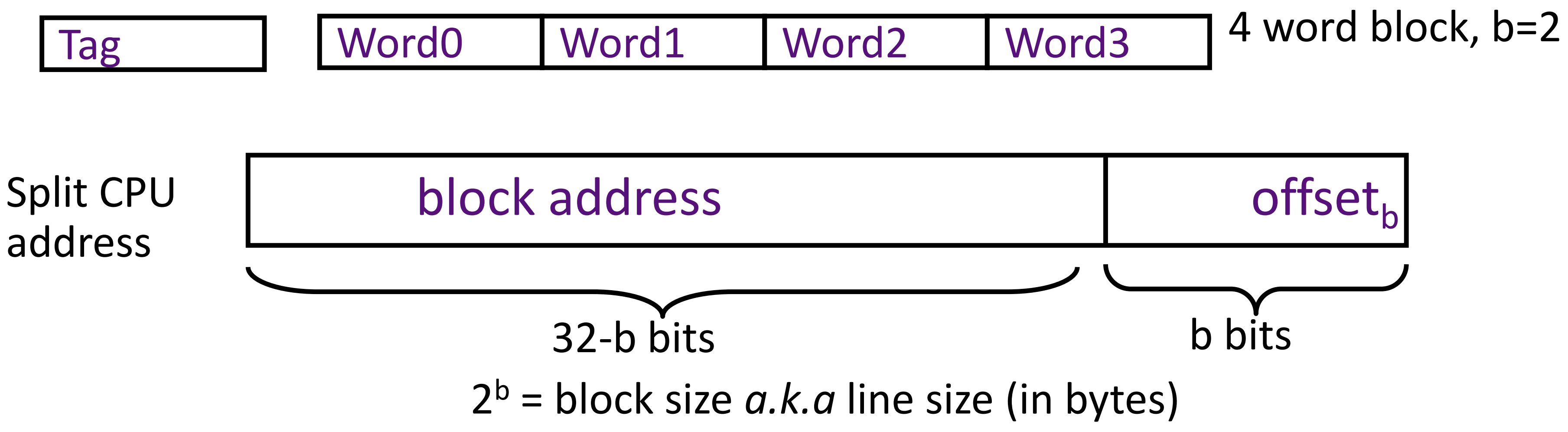
# What’s in Tag Store?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches



# Block (line) Size ?



$2^b$ = block size *a.k.a* line size (in bytes)

Larger block size has distinct hardware advantages

- less tag overhead
- exploit fast burst transfers from DRAM
- exploit fast burst transfers over wide busses

*What are the disadvantages of increasing block size?*

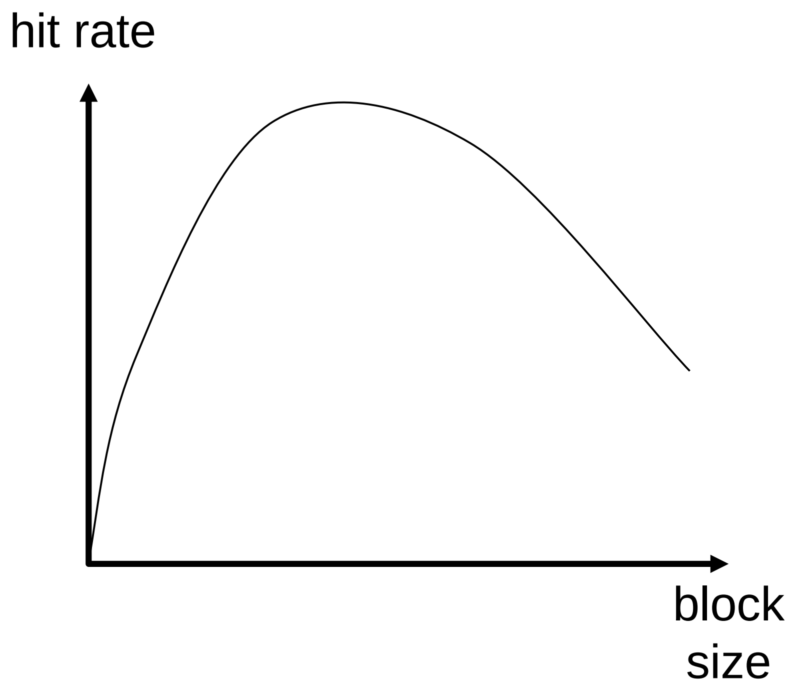
*Fewer blocks => more conflicts. Can waste bandwidth.*

## Block Size?

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies

- Too small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead

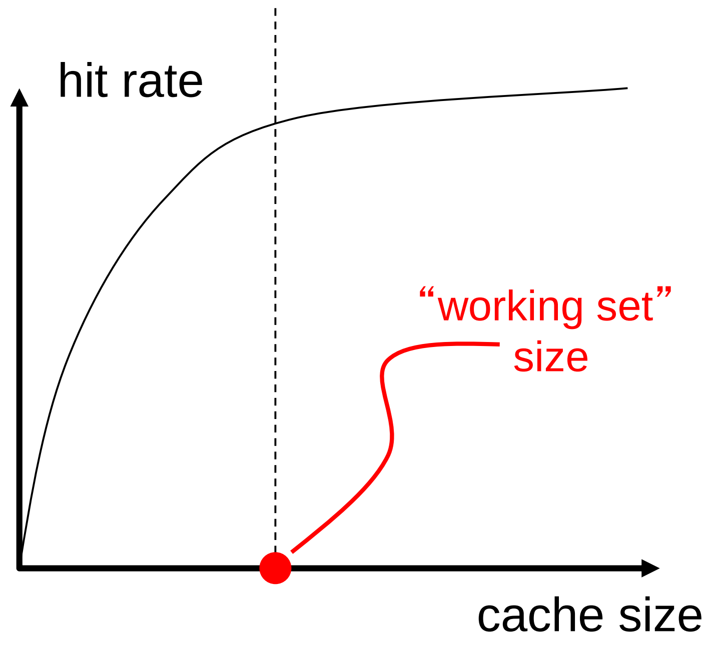
- Too large blocks
  - too few total # of blocks
    - likely-useless data transferred
    - Extra bandwidth/energy consumed



# Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn’t exploit temporal locality well
  - useful data replaced often

- Working set: the whole set of data the executing application references
  - Within a time interval



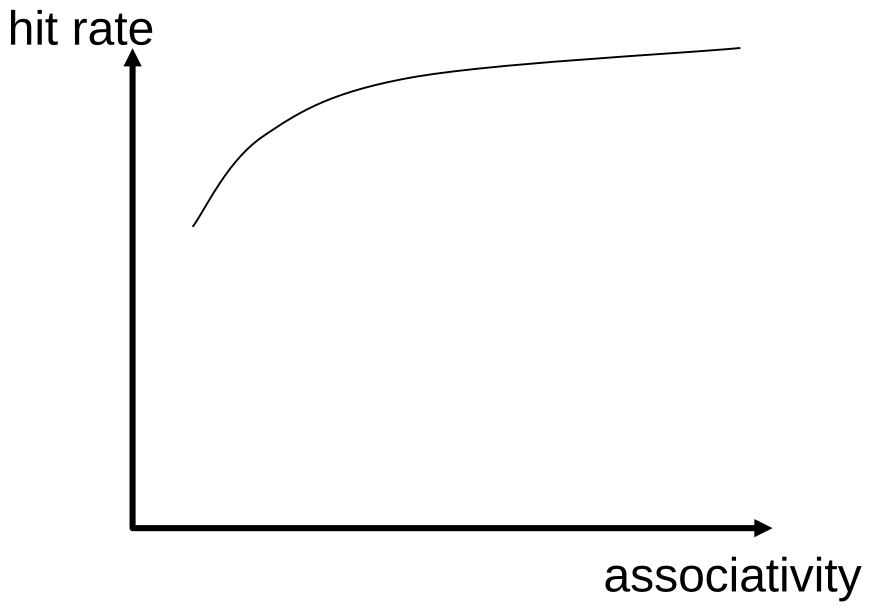
# Associativity

- How many blocks can map to the same index (or set)?

- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency

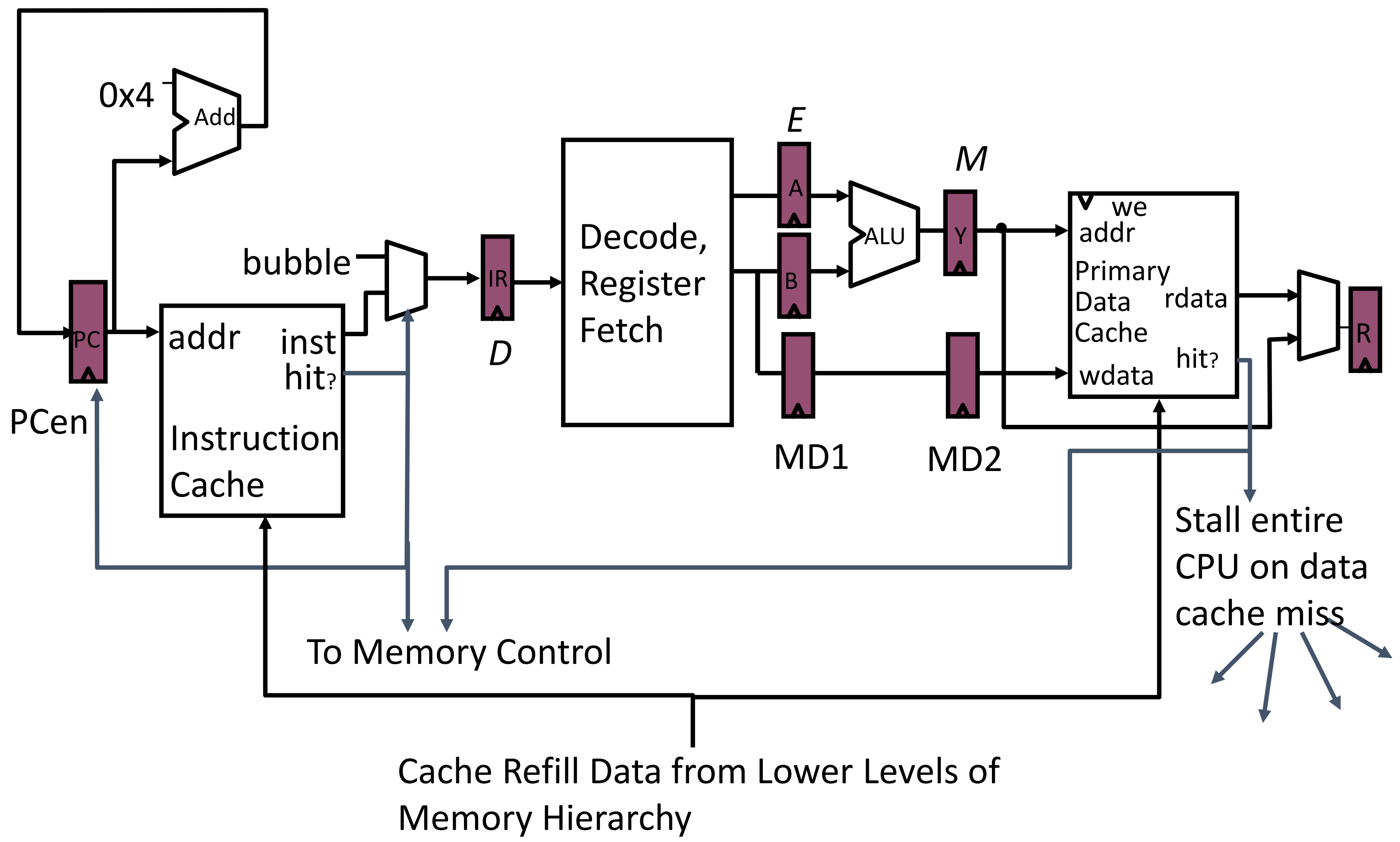
- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Power of 2 associativity?



## Let’s Pause!!

# CPU – Cache Interaction



## Design Issues: Unified vs Split

- Unified:

  + Dynamic sharing of cache space: no overprovisioning

  -- Instructions and data can thrash each other

  -- I and D are accessed in different places in the pipeline.

- First level caches are almost always split
  - for the reasons above

- Second and higher levels are almost always unified

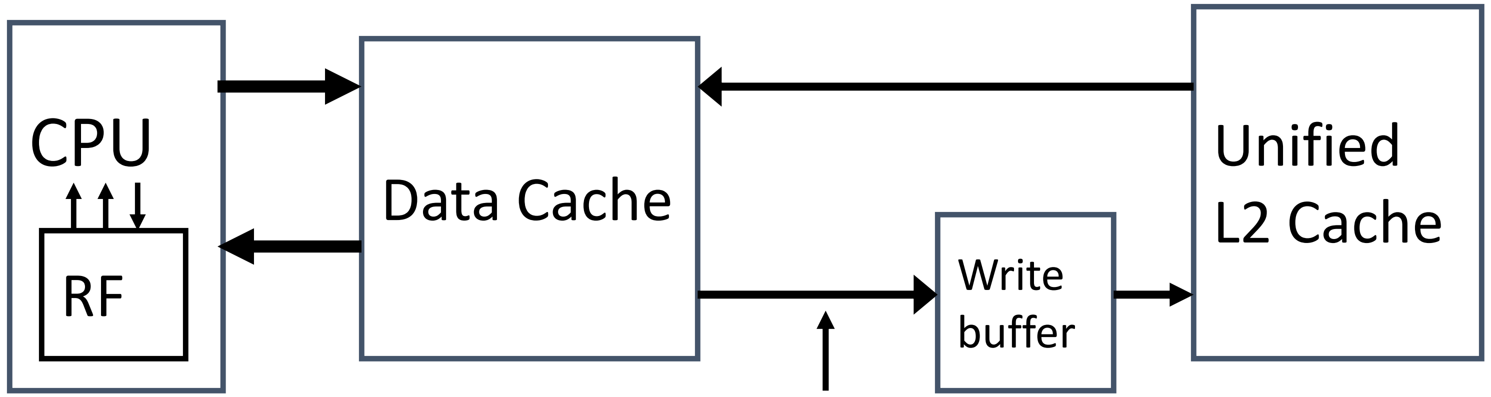
## Reads are not Writes

- If a write enters the cache, what happens if
  - There is a cache miss
    - Does the cache need to bring in the cache line?
  - There is a cache hit
    - Does the cache need to write back to memory?

## Write Policies

- Cache hit:
  - ***write through***: write both cache & memory
    - Generally higher traffic but simpler pipeline & cache design
  - ***write back***: write cache only, memory is written only when the entry is evicted
    - A dirty bit per line further reduces write-back traffic
    - Must handle 0, 1, or 2 accesses to memory for each load/store
- Cache miss:
  - ***no write allocate***: only write to main memory
  - ***write allocate*** (aka fetch on write): fetch into cache
- Common combinations:
  - write through and no write allocate
  - write back with write allocate

## Write Buffers



Evicted dirty lines for writeback cache
OR
All writes in writethrough cache

Processor is not stalled on writes, and read misses can go ahead of write to main memory

**Problem:** Write buffer may hold updated value of location needed by a read miss

**Simple solution:** on a read miss, wait for the write buffer to go empty

**Faster solution:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

## Performance: AMAT

Average memory access time (AMAT) = Hit time + Miss rate x Miss penalty

Average memory access time (AMAT) = Hit time + Miss rate$_1$ x Miss penalty$_1$
+ Miss rate$_2$ x Miss penalty$_2$

# Improving Cache Performance

Average memory access time (AMAT) =
Hit time + Miss rate x Miss penalty

To improve performance:

- reduce the hit time
- reduce the miss rate
- reduce the miss penalty

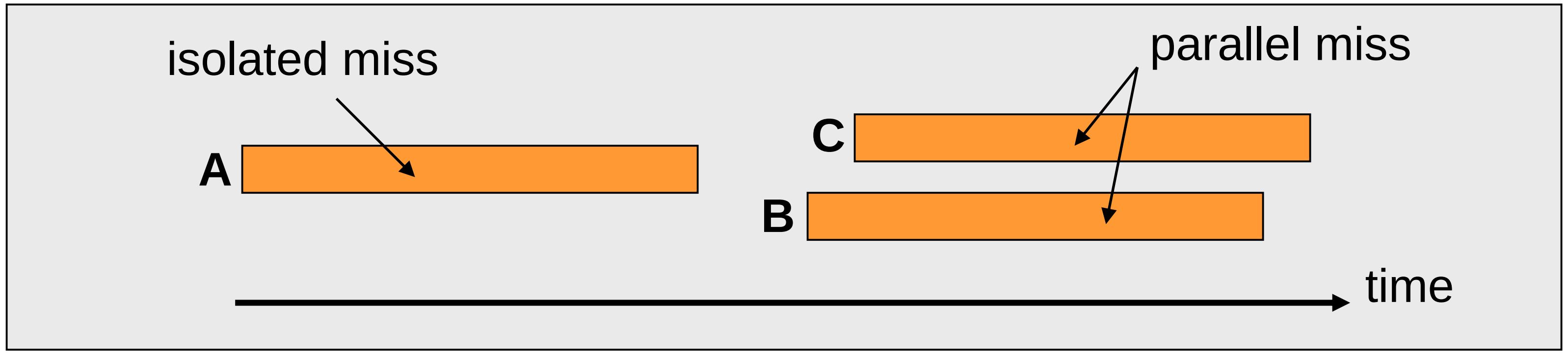
*Biggest cache that doesn't increase hit time past 1 cycle (approx 8-32KB in modern technology)*

*[ design issues more complex with deeper pipelines and/or out-of-order superscalar processors]*

## Cache Optimizations: Refer H&P

# Non-blocking Cache

- Enable cache access when there is a pending miss
- Enable multiple misses in parallel
    - Memory-level parallelism (MLP)
        - generating and servicing multiple memory accesses in parallel
    - Why generate multiple misses?



        - Enables latency tolerance: overlaps latency of different misses
    - How to generate multiple misses?
        - Out-of-order execution, multithreading, prefetching

## Miss-Status Holding Registers

- Also called “miss buffer”
- Keeps track of
  - Outstanding cache misses
  - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR
  - Valid bit
  - Cache block address (to match incoming accesses)
  - Control/status bits (prefetch, issued to memory, which subblocks have arrived, etc)
  - Data for each subblock
  - For each pending load/store
    - Valid, type, data size, byte in block, destination register or store buffer entry address

# MSHRs

| 1 | 27 | 1 |
|---|---|---|
| Valid | Block Address | Issued |

| 1 | 3 | 5 | 5 | |
|---|---|---|---|---|
| Valid | Type | Block Offset | Destination | Load/store 0 |
| Valid | Type | Block Offset | Destination | Load/store 1 |
| Valid | Type | Block Offset | Destination | Load/store 2 |
| Valid | Type | Block Offset | Destination | Load/store 3 |

## MSHR in Action

- On a cache miss:
  - Search MSHR for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall

- When a subblock returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - Deallocate load/store entry in the MSHR entry
  - Write subblock in cache or MSHR
  - If last subblock, dellaocate MSHR (after writing the block in cache)

# The 3Cs

**Compulsory:**

first reference to a line (a.k.a. cold start misses)

- *misses that would occur even with infinite cache*

**Capacity:**

cache is too small to hold all data needed by the program

- *misses that would occur even under perfect replacement policy*

**Conflict:**

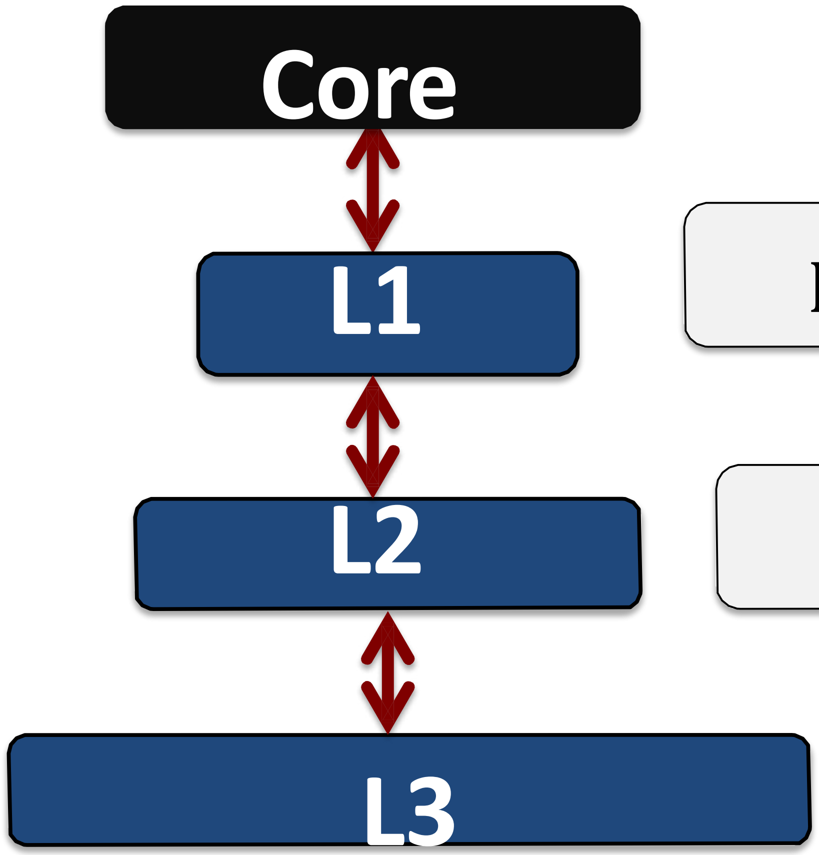
misses that occur because of collisions due to line-placement strategy

- *misses that would not occur with ideal full associativity*

## Cache Knobs and Performance

- Larger cache size
  - + reduces capacity and conflict misses
  - - hit time will increase
- Higher associativity
  - + reduces conflict misses
  - - may increase hit time
- Larger line size
  - + reduces compulsory and capacity (reload) misses
  - - increases conflict misses and miss penalty

## Cache Hierarchy

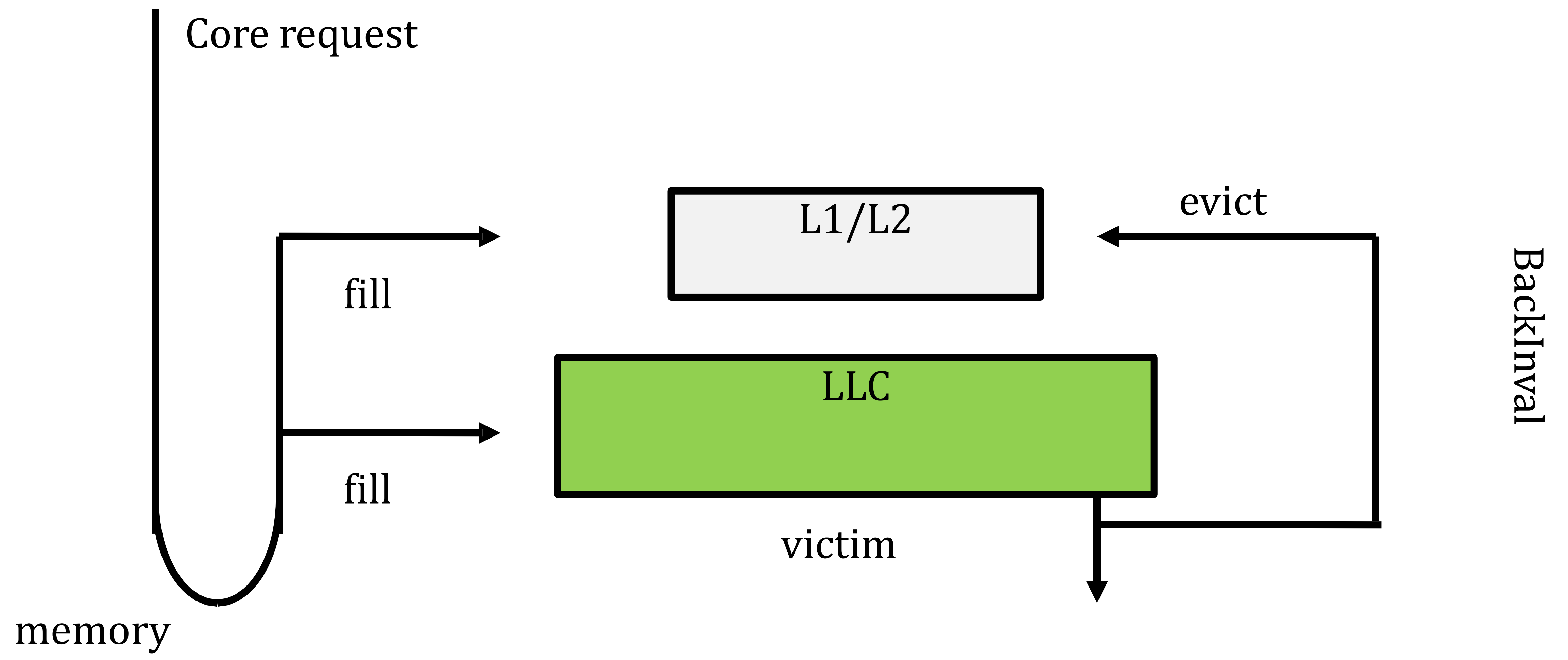


Designed for *bandwidth*

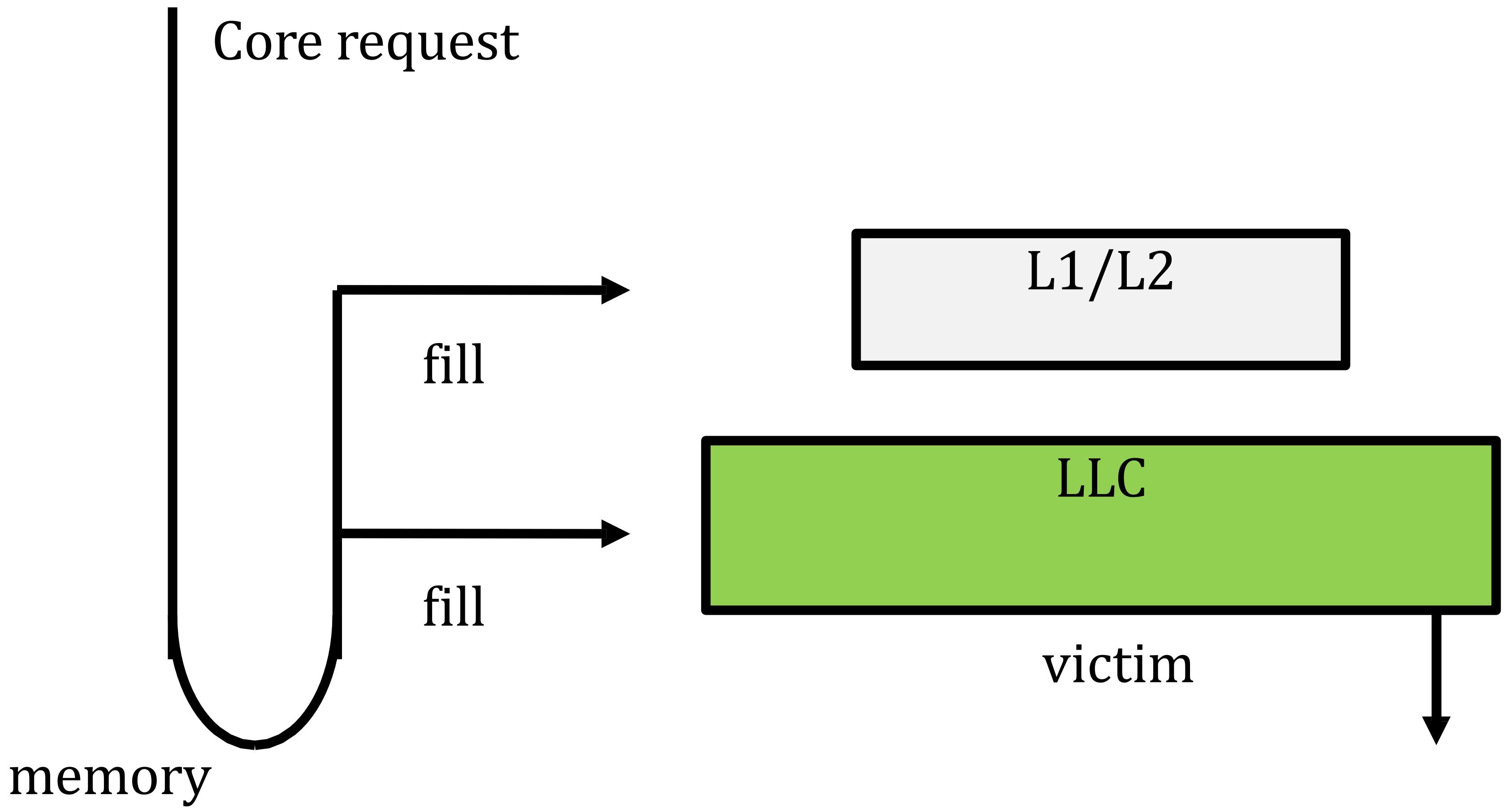
Designed for *latency*

Designed for *capacity*

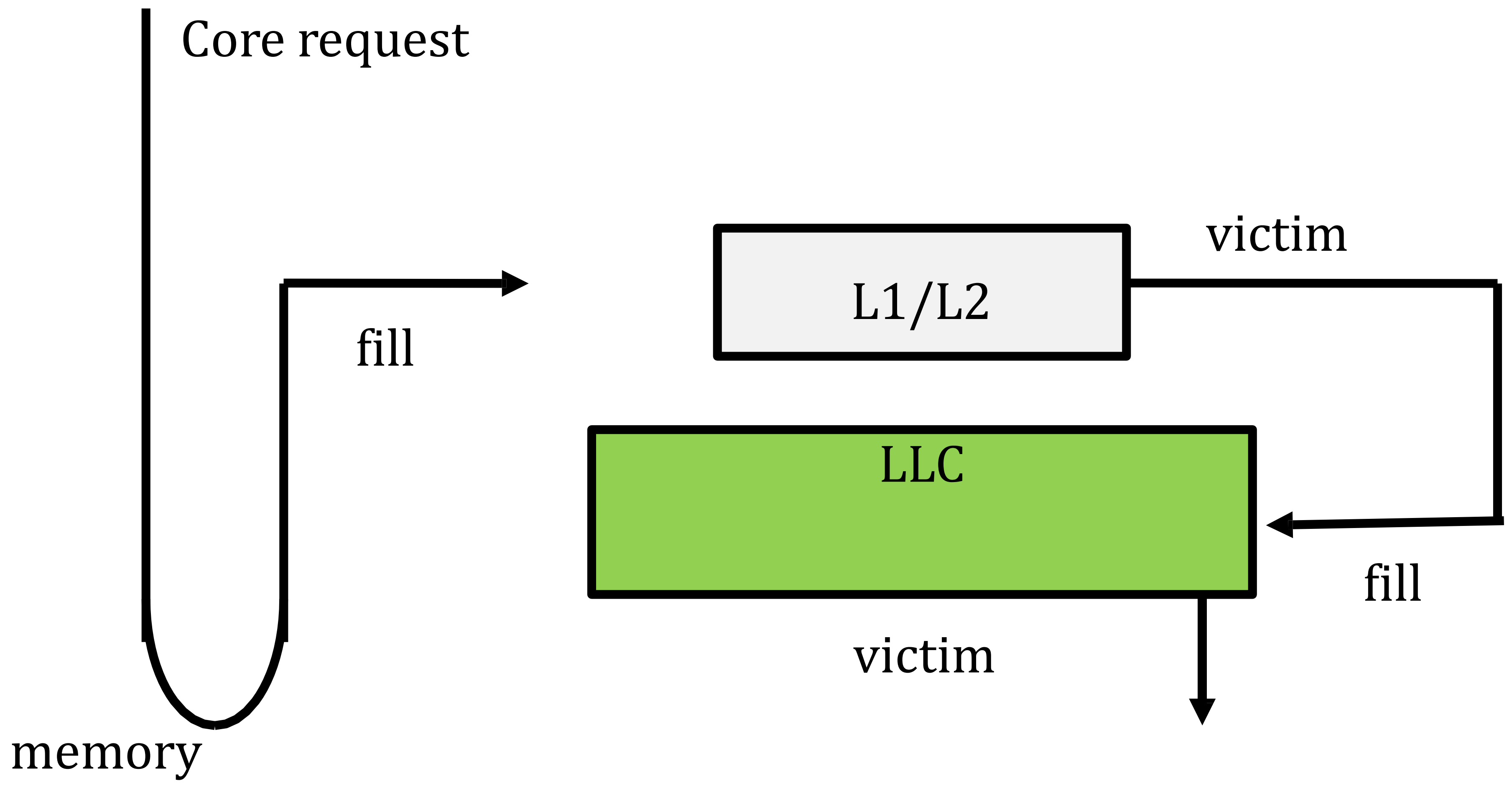
## Inclusive


Core request
L1/L2
evict
fill
BackInval
LLC
fill
victim
memory

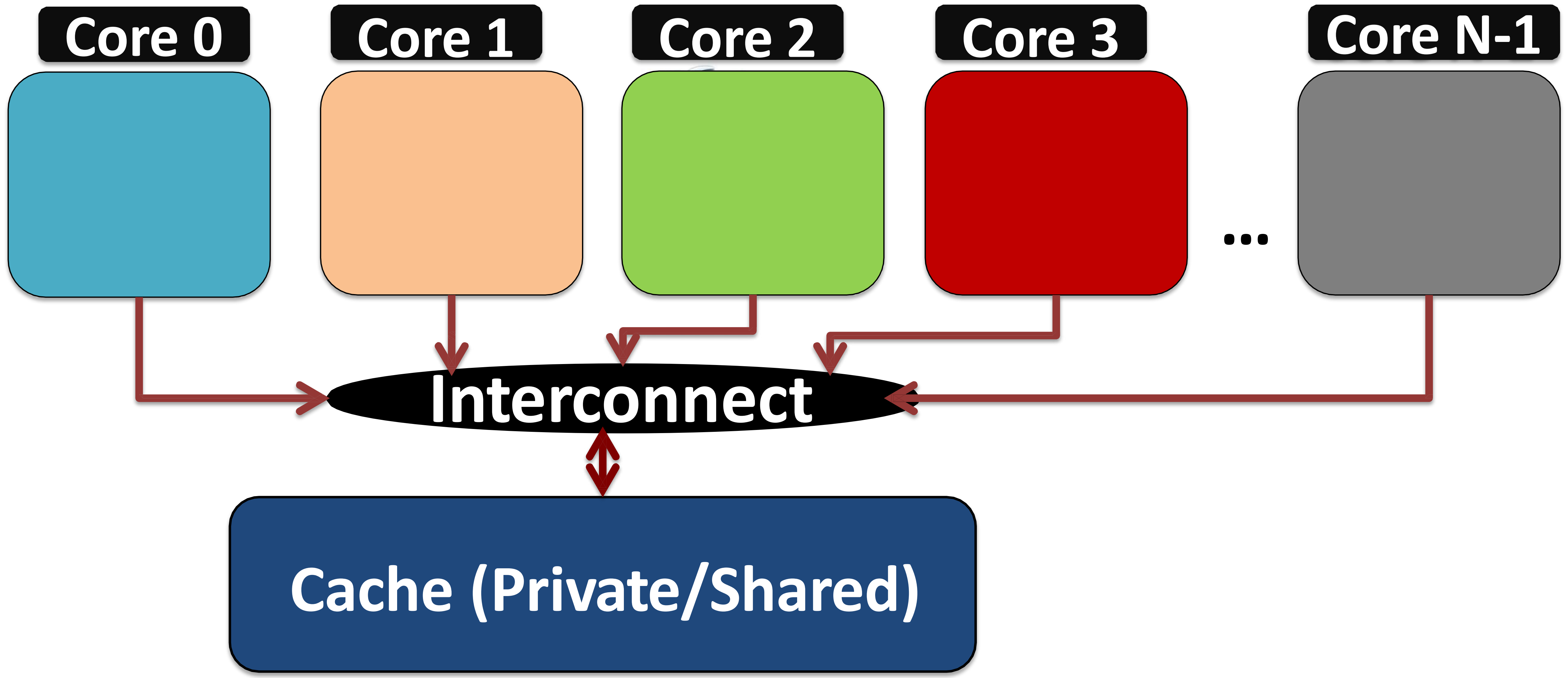
# Non-inclusive



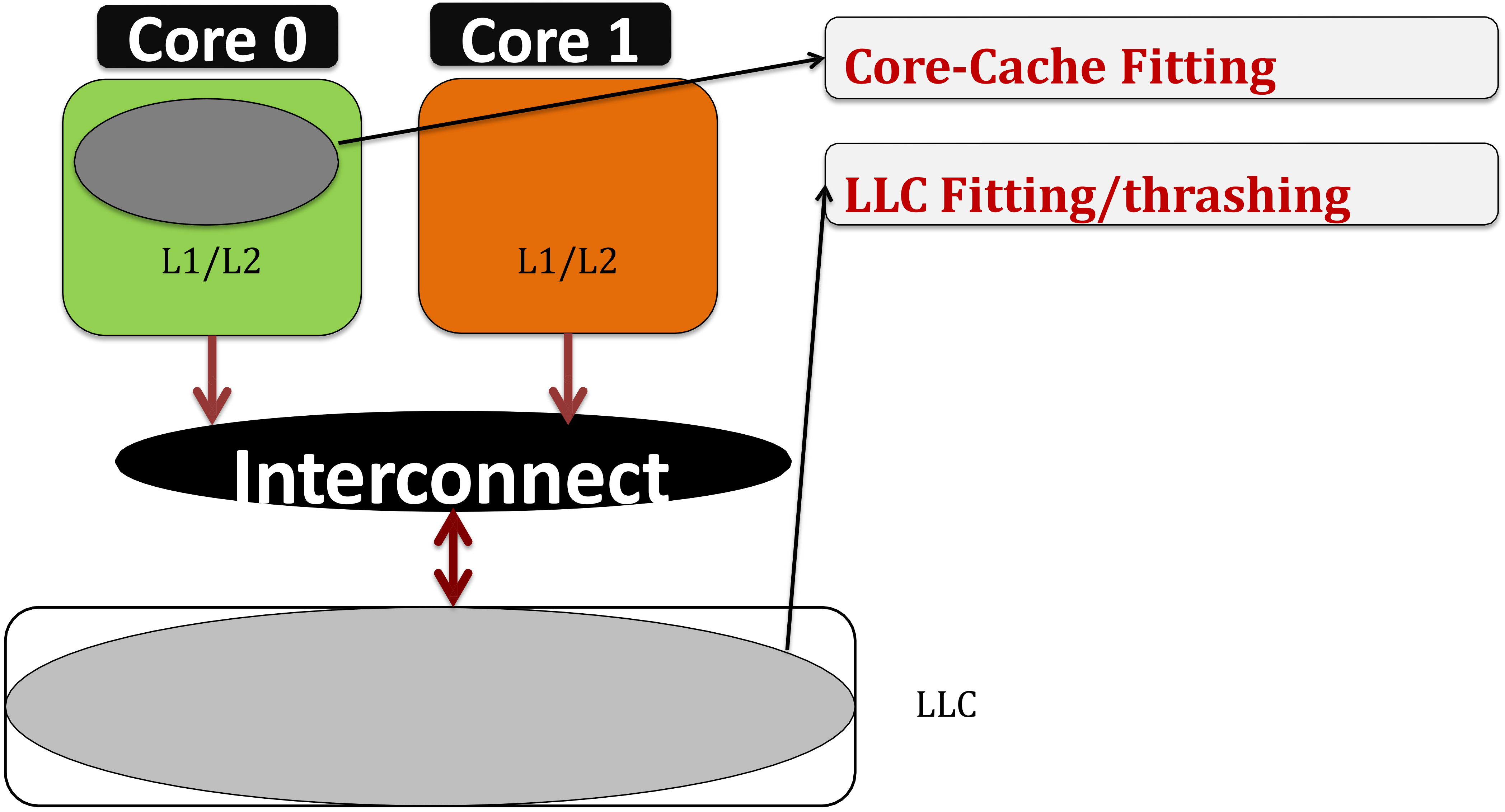
# Exclusive



# LLC: Shared or Private?



# Application Behavior



## Shared LLC

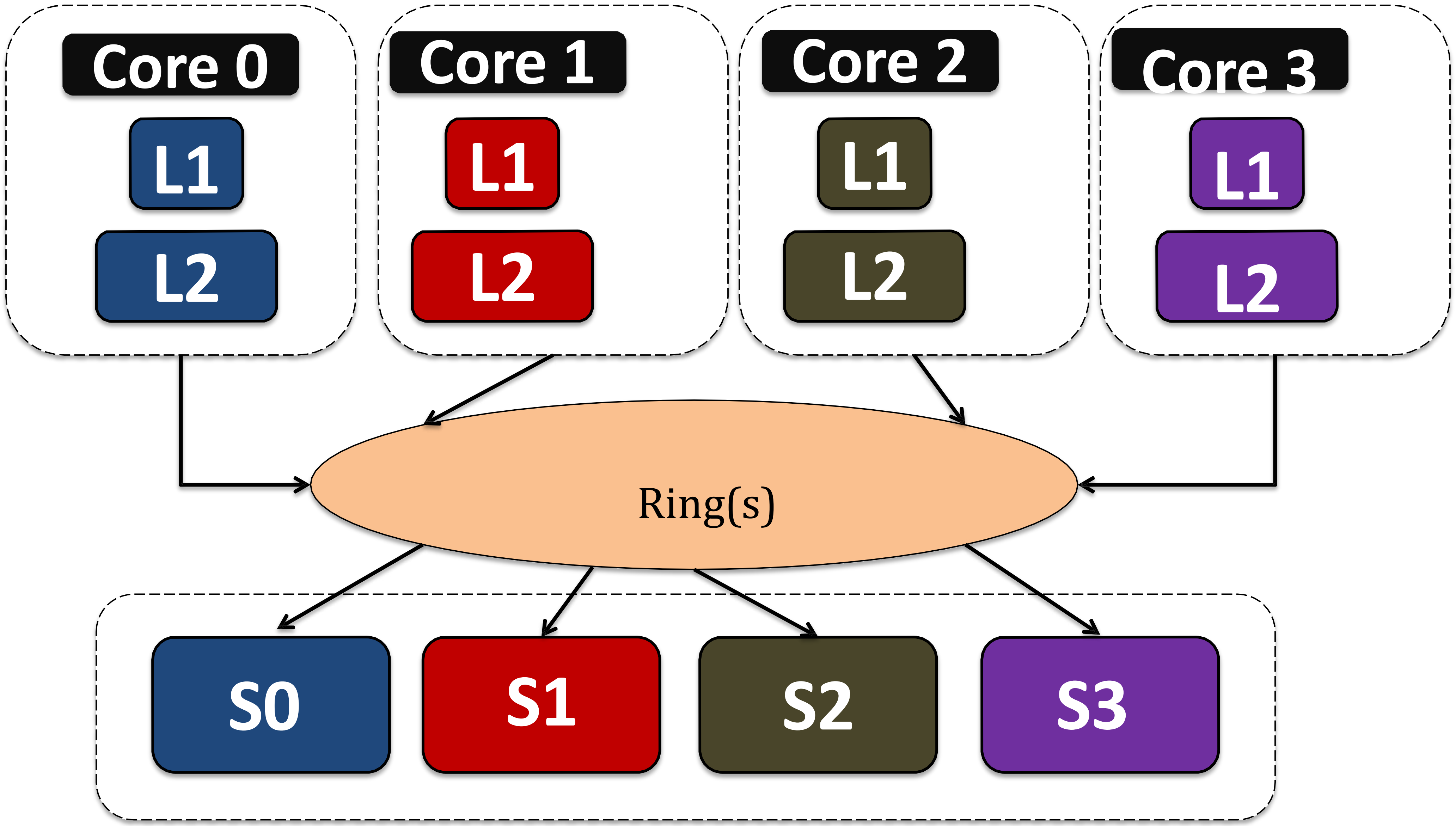
Shared LLC provides a good tradeoff for all kinds of apps.

Space unutilized by one app. can be utilized by other apps.

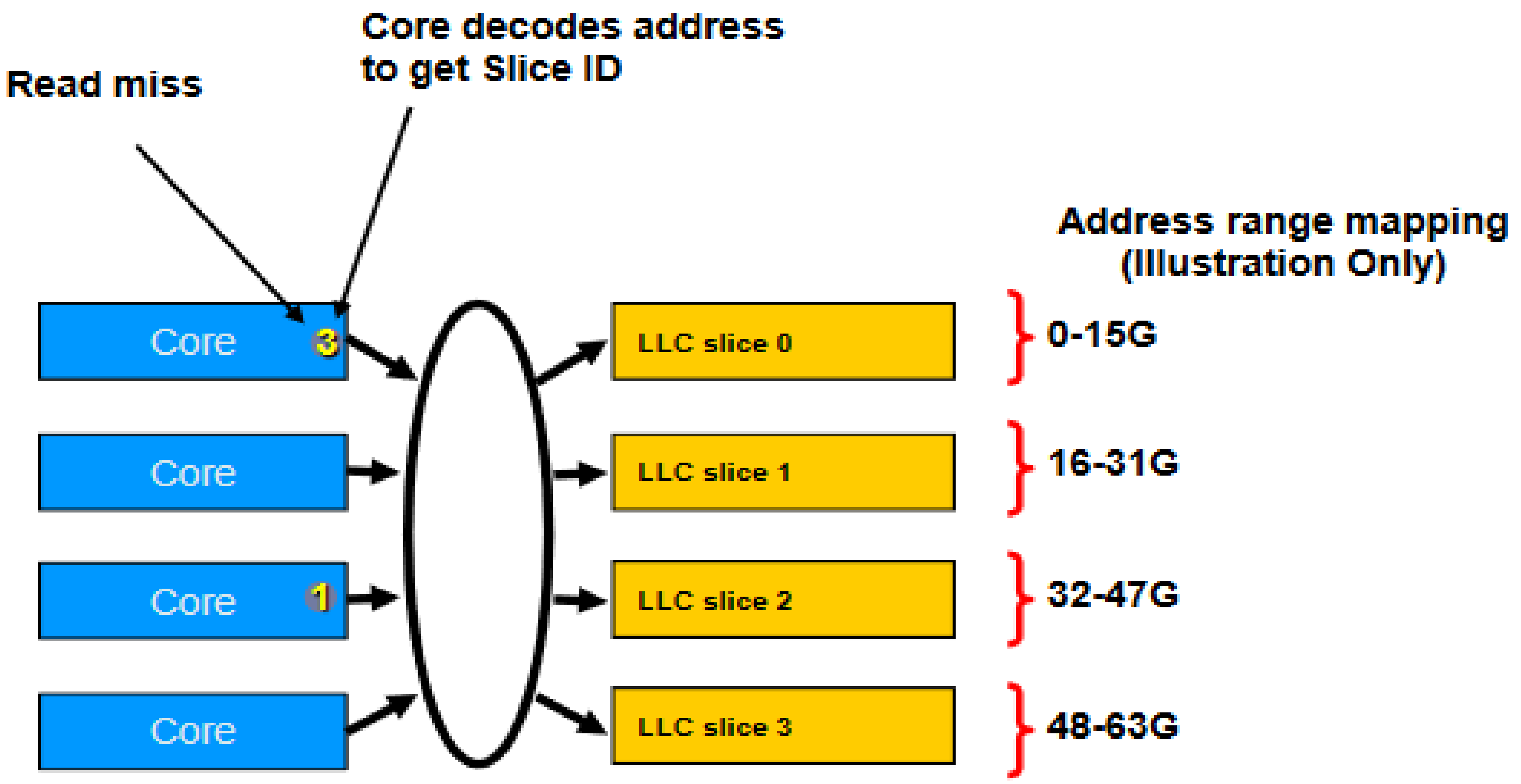
However bandwidth is an issue ☹

1000 monkeys: one banana

## Banked/Sliced NUCA



# Intel’s Sandybridge



# Cache Replacement-101

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

# Eviction (Replacement) Policy?

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

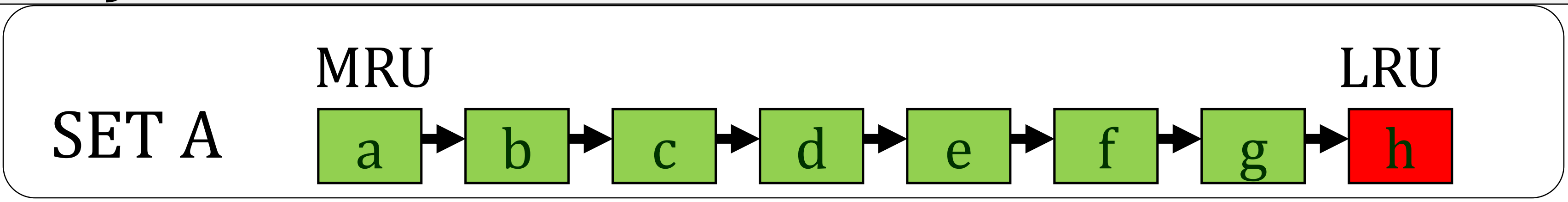
# Belady

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

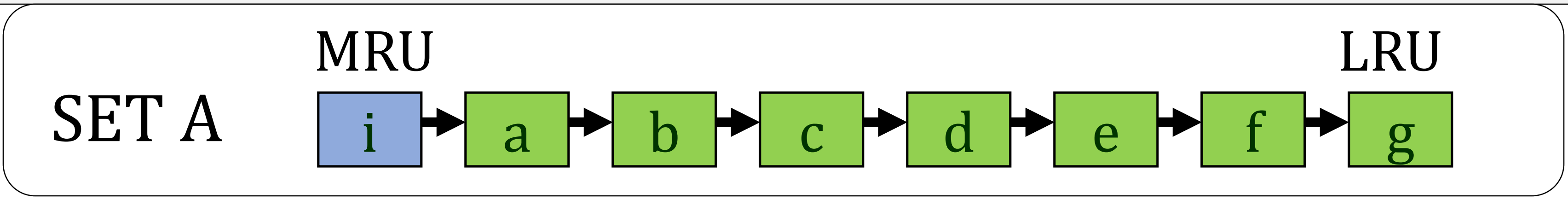
- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

# LRU - 101

**Cache Eviction Policy: On a miss (block *i*), which block to evict (replace) ?**



**Cache Insertion Policy: New block *i* inserted into MRU.**



**Cache Promotion Policy: On a future hit (block *i*), promote to MRU**

**LRU causes thrashing when working set > cache size**

## Access Patterns

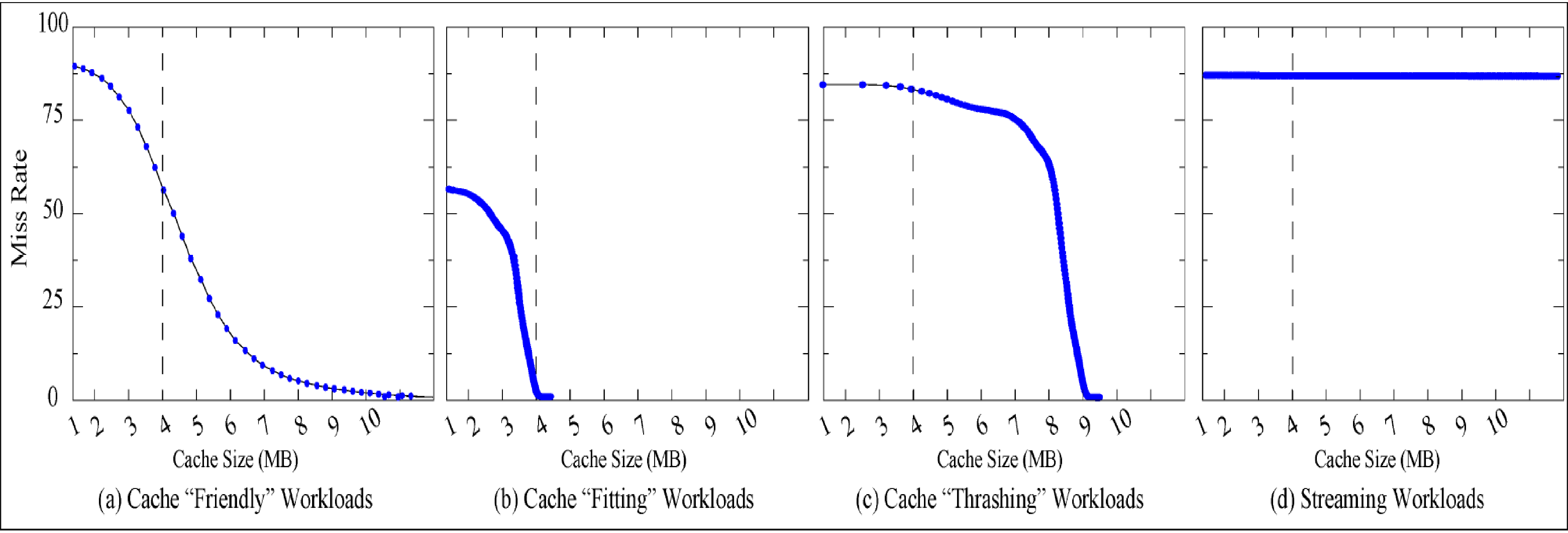
**Recency friendly** $(a_1, a_2, \ldots a_k, a_{k-1}, \ldots a_2, a_1)^N$

**Thrashing** $(a_1, a_2, \ldots a_k)^N$ [k > cache size]

**Streaming** $(a_1, a_2, \ldots a_\infty)^N$

**Combination of above three**

# Types of Workloads – 4MB cache



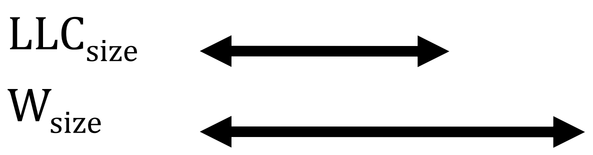
## Limitations of LRU

**LRU exploits temporal locality**

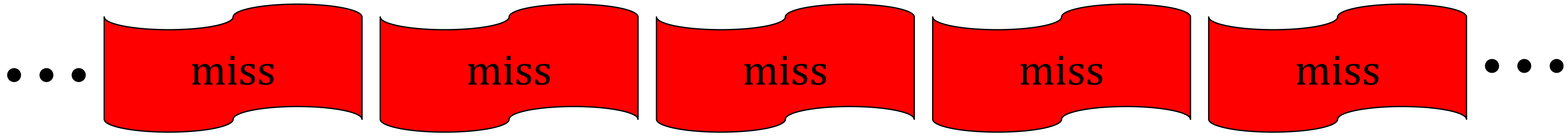
**Streaming data ($a_1$, $a_2$, $a_3$,....$a_\infty$):**
**No temporal locality,**
**No temporal reuse**

**Thrashing data ($a_1$, $a_2$, $a_3$,....$a_n$) [$n>c$]**
**Temporal locality exists. However, LRU fails to capture.**

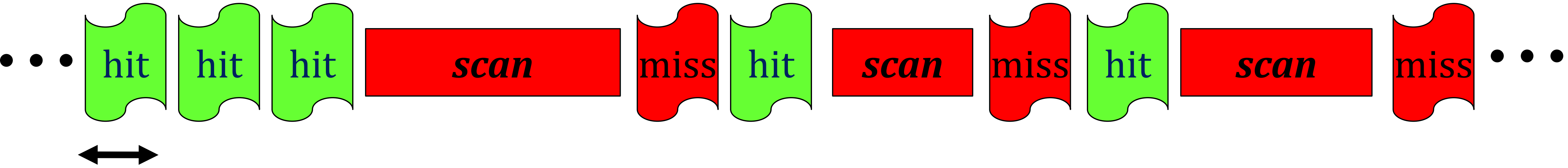
# Still Miles to Go



Working set larger than the cache causes thrashing

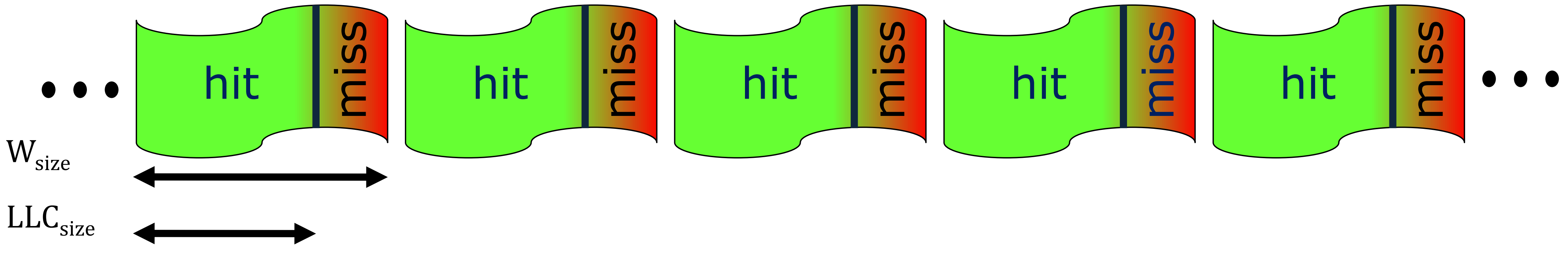


References to non-temporal data (*scans*) discards frequently referenced working set

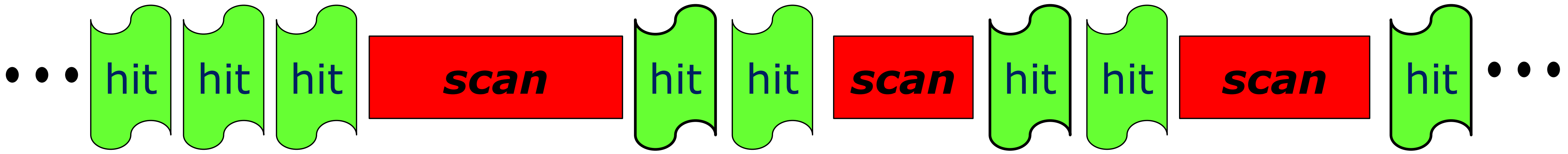


# Still Miles to Go

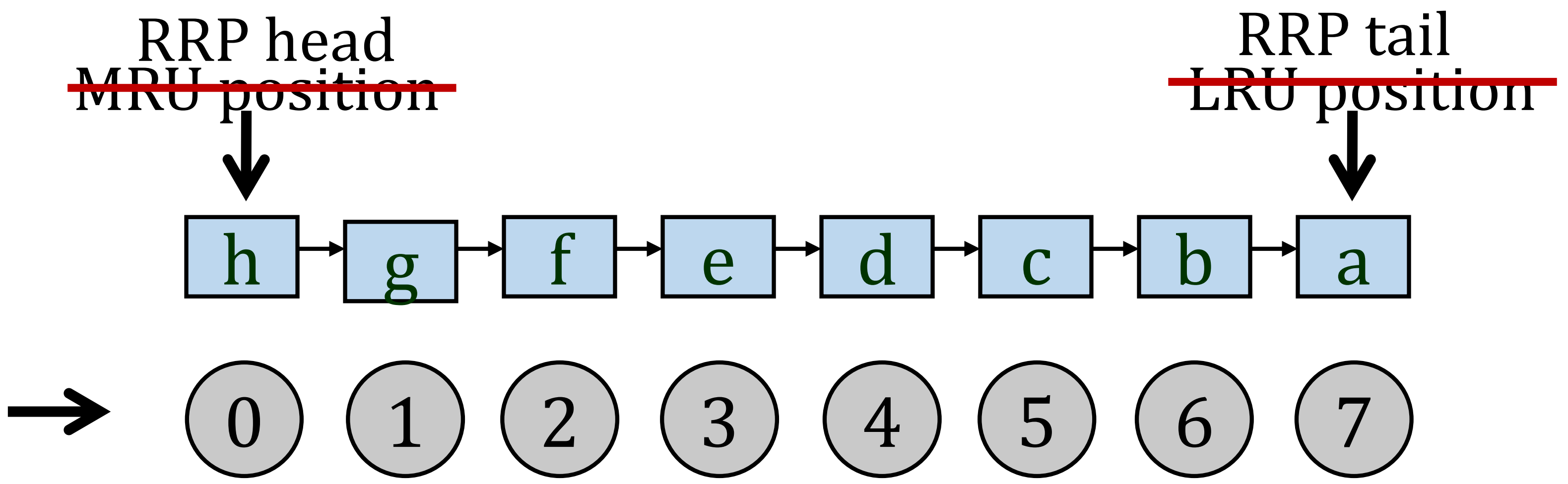
Working set larger than the cache ➔
Preserve some of working set in the cache



Recurring *scans (bursts of non-temporal data)* ➔ Preserve frequently referenced working set in the cache



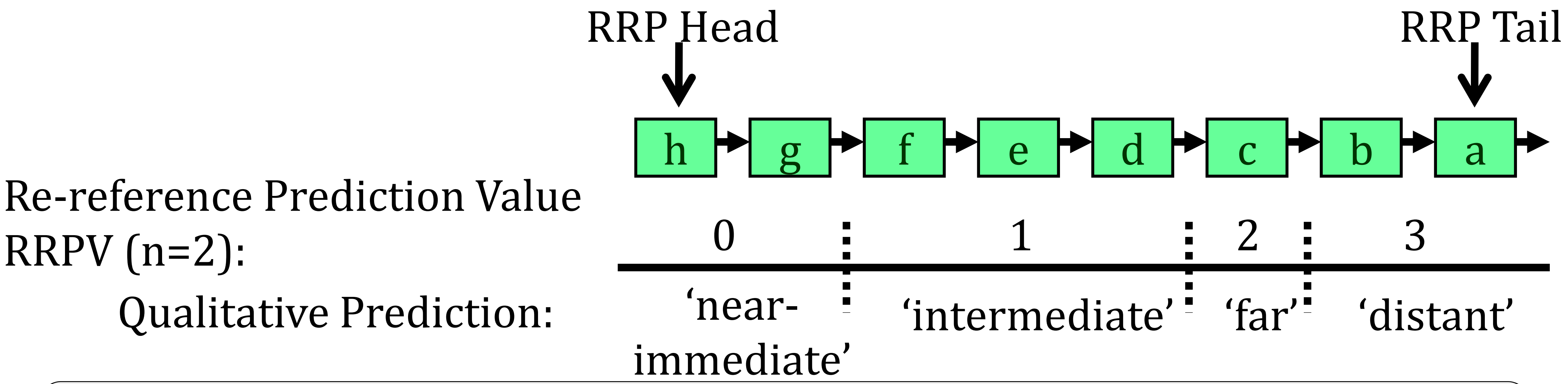
# RRIP – ISCA ‘10



LRU chain position stored with each cache block

RRP: Re-reference prediction

# RRIP

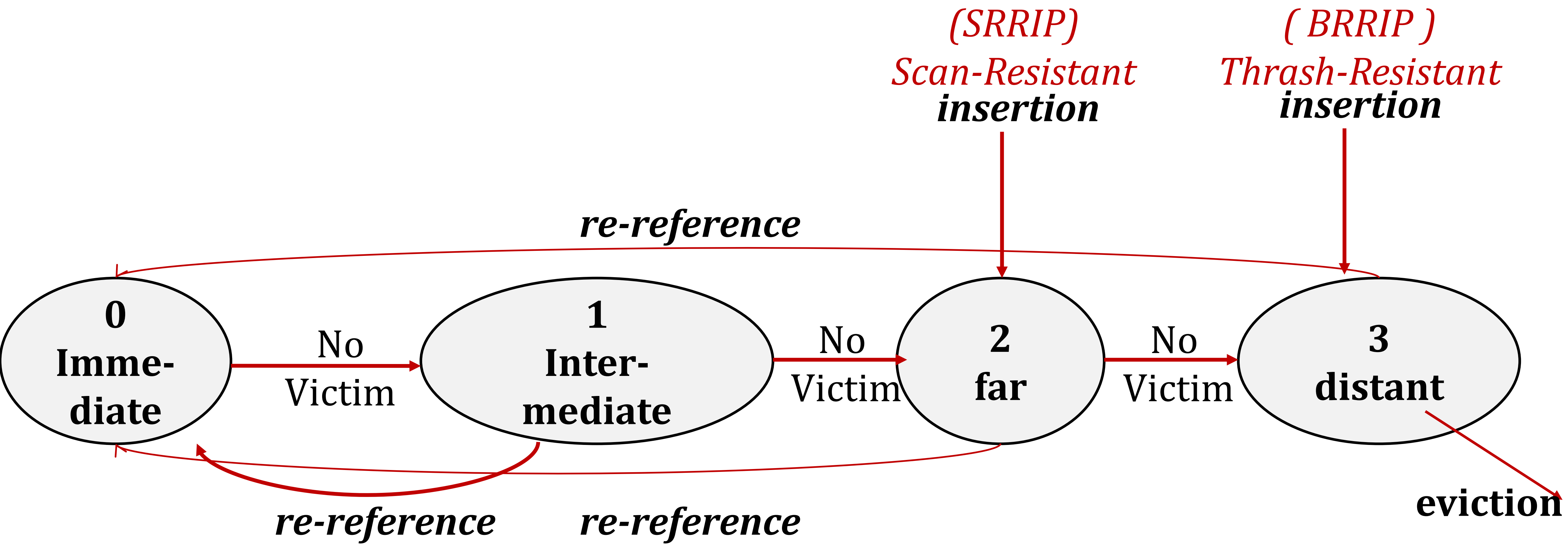


Intuition: New cache block will not be re-referenced soon. Replaces block with distant RRPV.

Insert with RRPV=2, Evict with RRPV=3
promote blocks with RRPV=0.

Static RRIP (Single core) and Thread-Aware Dynamic RRIP (SRRIP+BRRIP, multi-core, based on SDMs).

# DRRIP


(SRRIP)
Scan-Resistant
insertion
( BRRIP )
Thrash-Resistant
insertion
re-reference
0
Imme-
diate
No
Victim
1
Inter-
mediate
No
Victim
2
far
No
Victim
3
distant
eviction
re-reference
re-reference

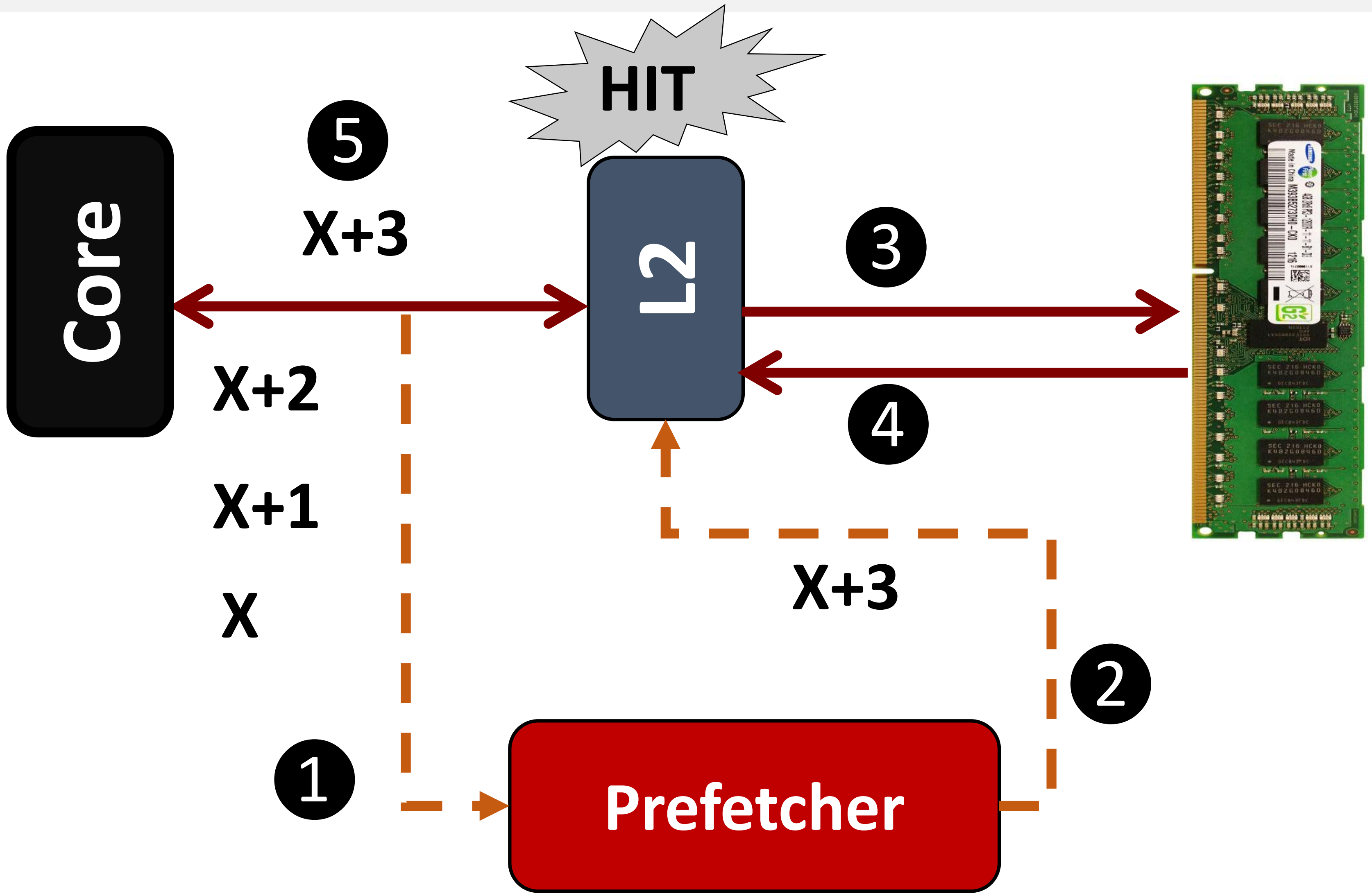
# Hardware Prefetching

*What?*
Latency-hiding technique - Fetches data before the core demands.

*Why?*
Off-chip DRAM latency has grown up to 400 to 800 cycles.

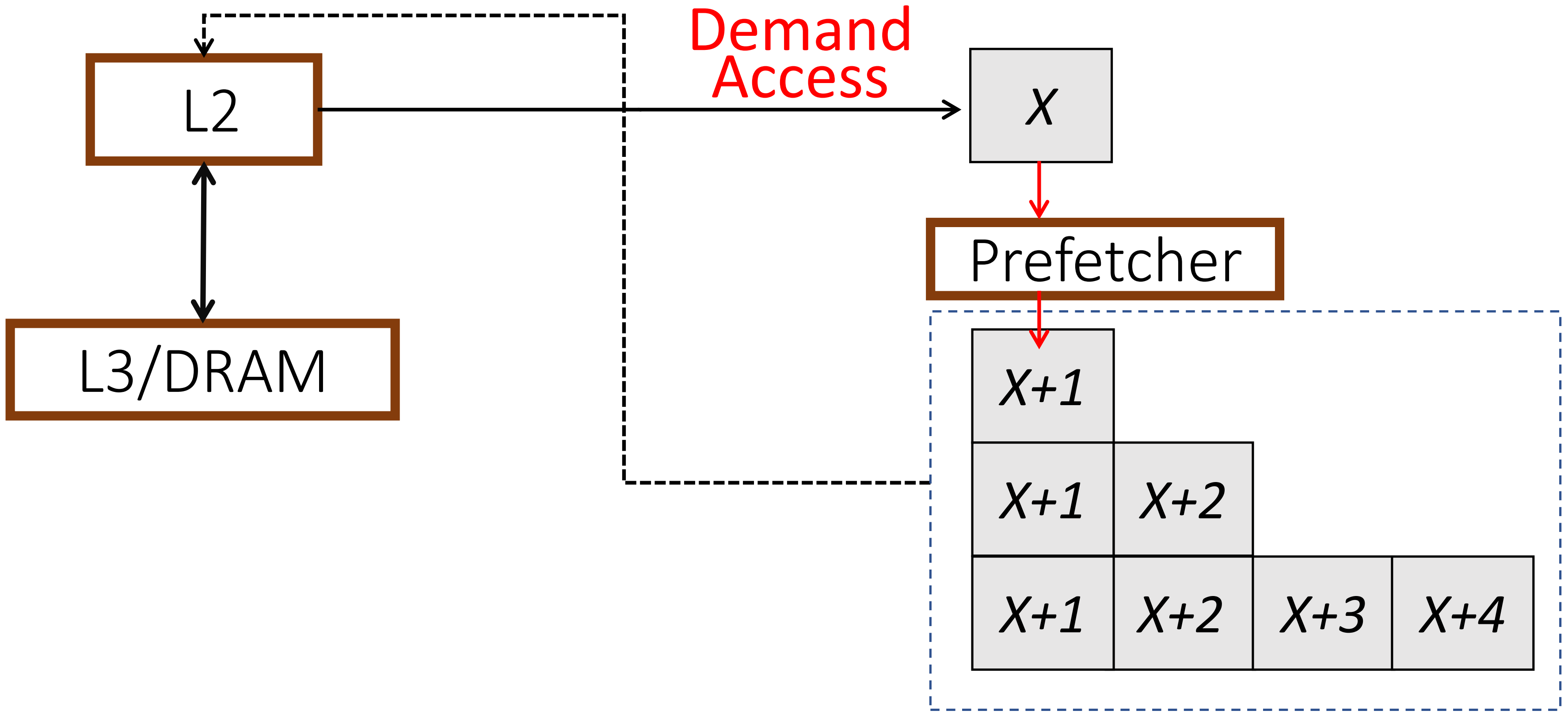
*How?*
By observing/predicting the demand access (LOAD/STORE) patterns.

# Prefetch Engine



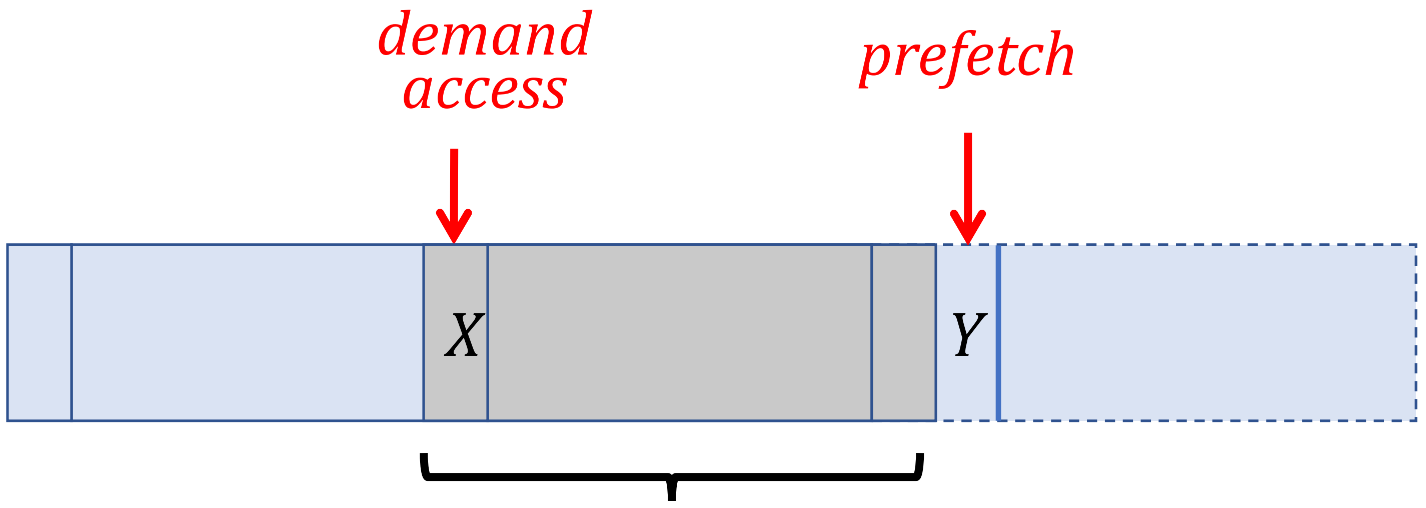
# Prefetch Degree

Prefetch Degree: Number of prefetch requests to issue at a given time.



# Prefetch Distance

**Prefetch Distance:** How far ahead of the demand access stream are the prefetch requests issued?



*Prefetch-distance*

$Y = X + 4$

$Y = X + 8$

$Y = X + 16$

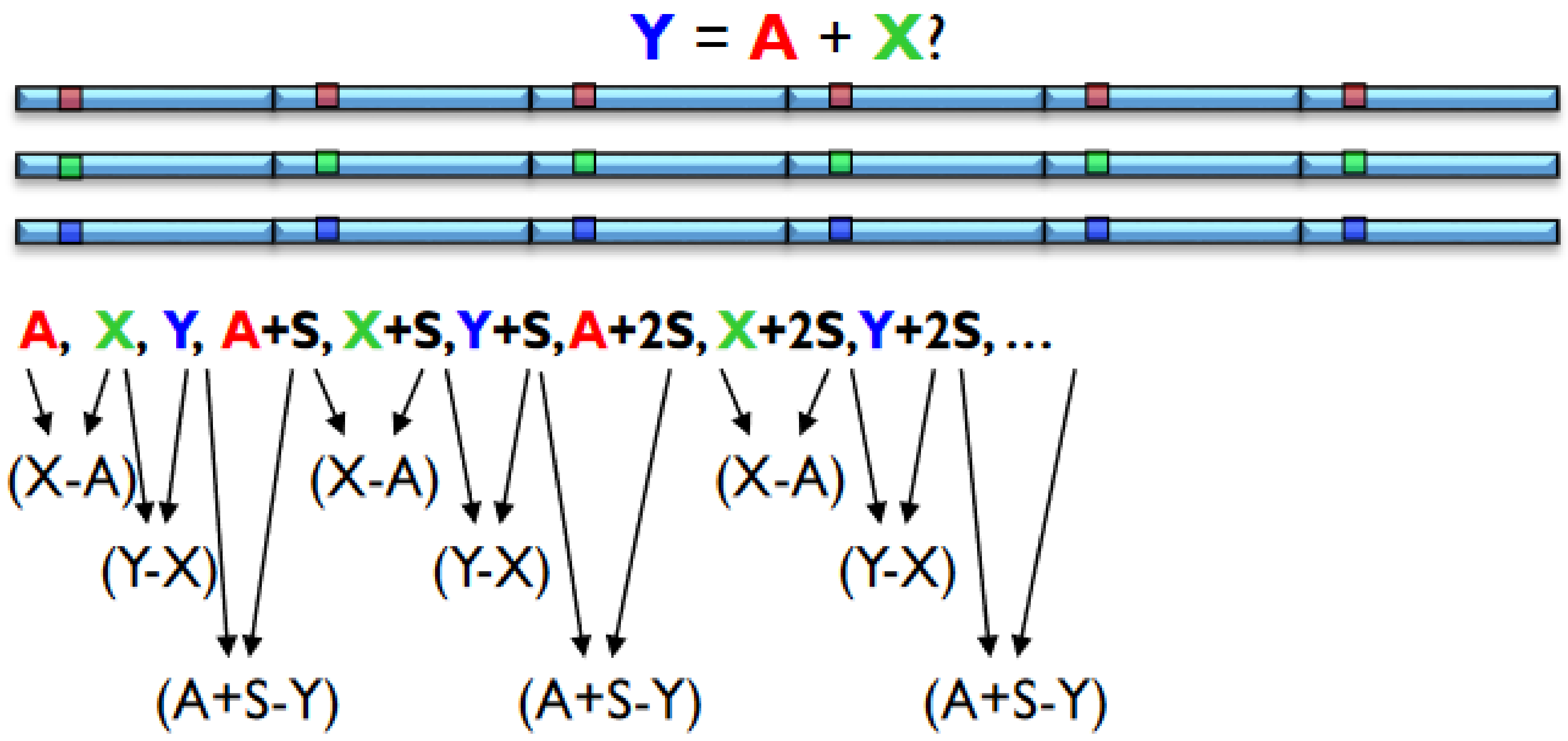
## Next-line Prefetcher

Next Line: Miss to cache block X , prefetch X+1. Degree=1, Distance=1

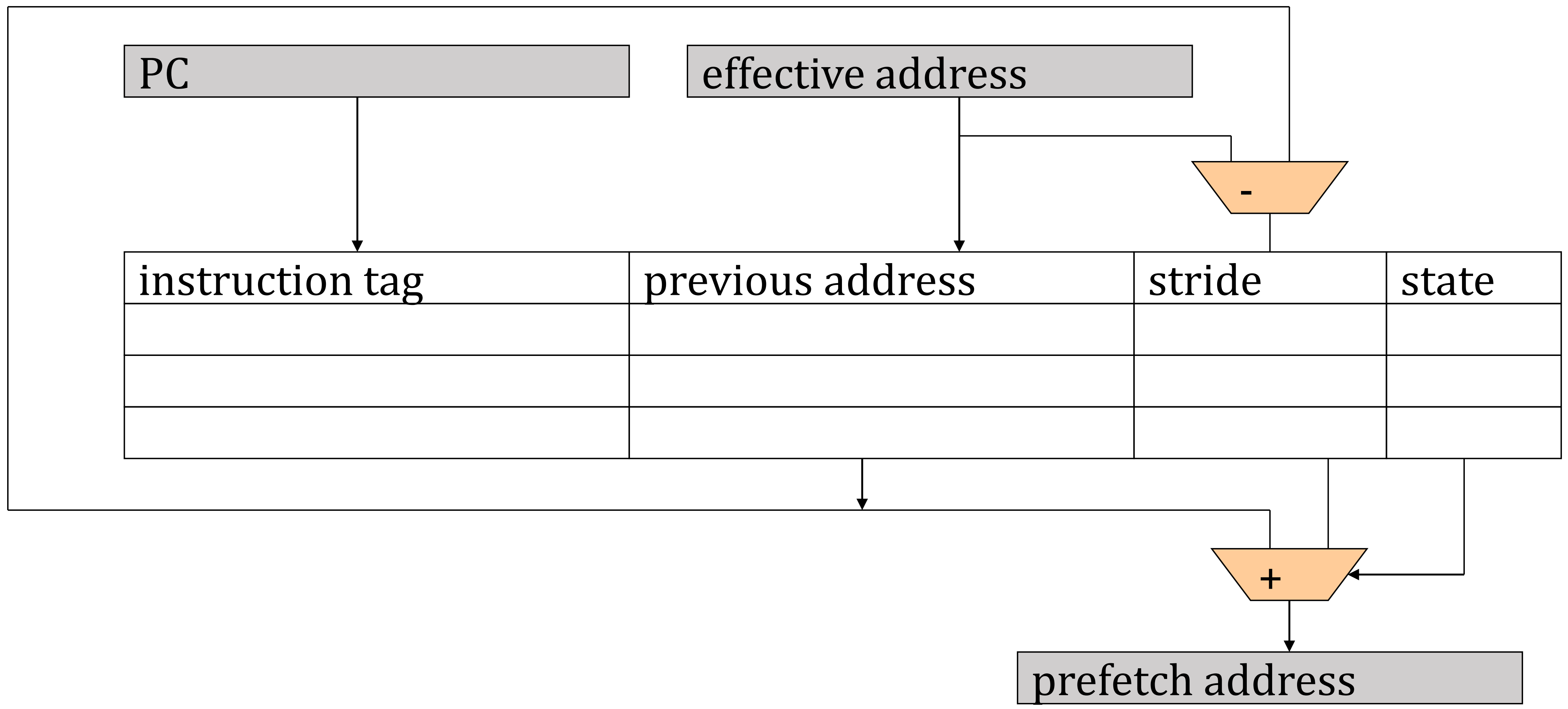
Works well for L1 Icache and L1 Dcache.

# What About This?

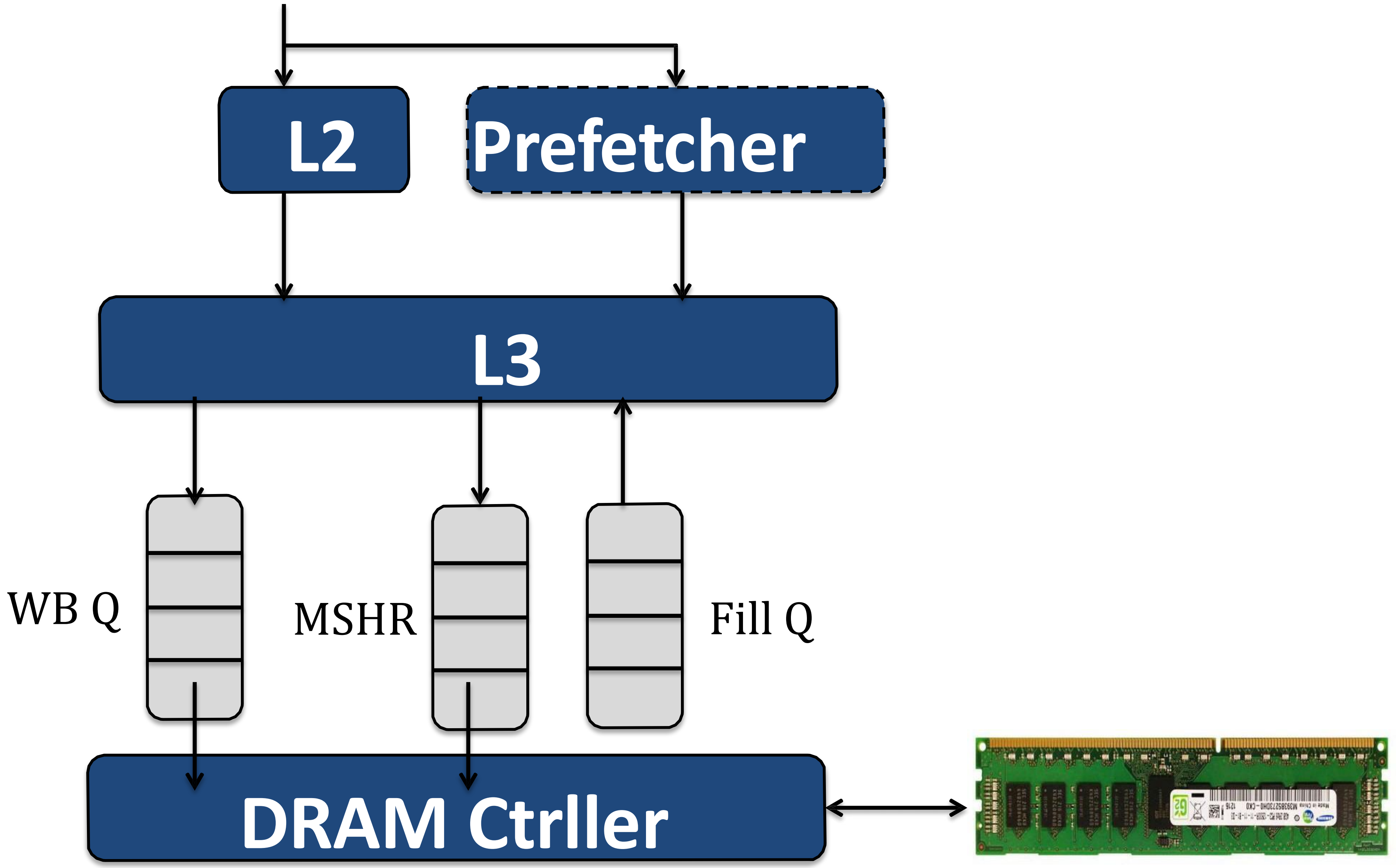
```
Load R1 = [R2]
Load R3 = [R4]
Add R5, R1, R3
Store [R6] = R5
```



# Stride Prefetching



# Bit Detailed



## ACKS

- Some of the slides are adapted and modified from Joel Emer, Arvind, Yale Patt, John Kubiatowicz, Onur Mutlu, Krste Asanovic, Mattan Erez, Rajeev Balasubramonian, and Mainak Chaudhuri