Lecture-2 & 3 (Brushing up the Processor)
CS665-Fall 2018
Secure Memory Systems

Biswa@CSE-IITK

Your Assignment-0: Congrats, you have scored a 0 ☺

Nuka: I promise then what ?
Supriya: I that promise I will try ? What does it mean
Newton: Do we have to call Biswa outside the class also?
Dixit: Submitted response twice ☺
@Few: Did not submit ☹

Riya: *to understand better when I read about security exploits in the news; to protect things I design as best as I can; to be able to read research papers more efficiently; to gain a better understanding of the intricacies of memory hierarchy.*

Logistics

Late submission: -10% for every 24 hours

Some of you are not in Piazza or not in Pingala or not in class

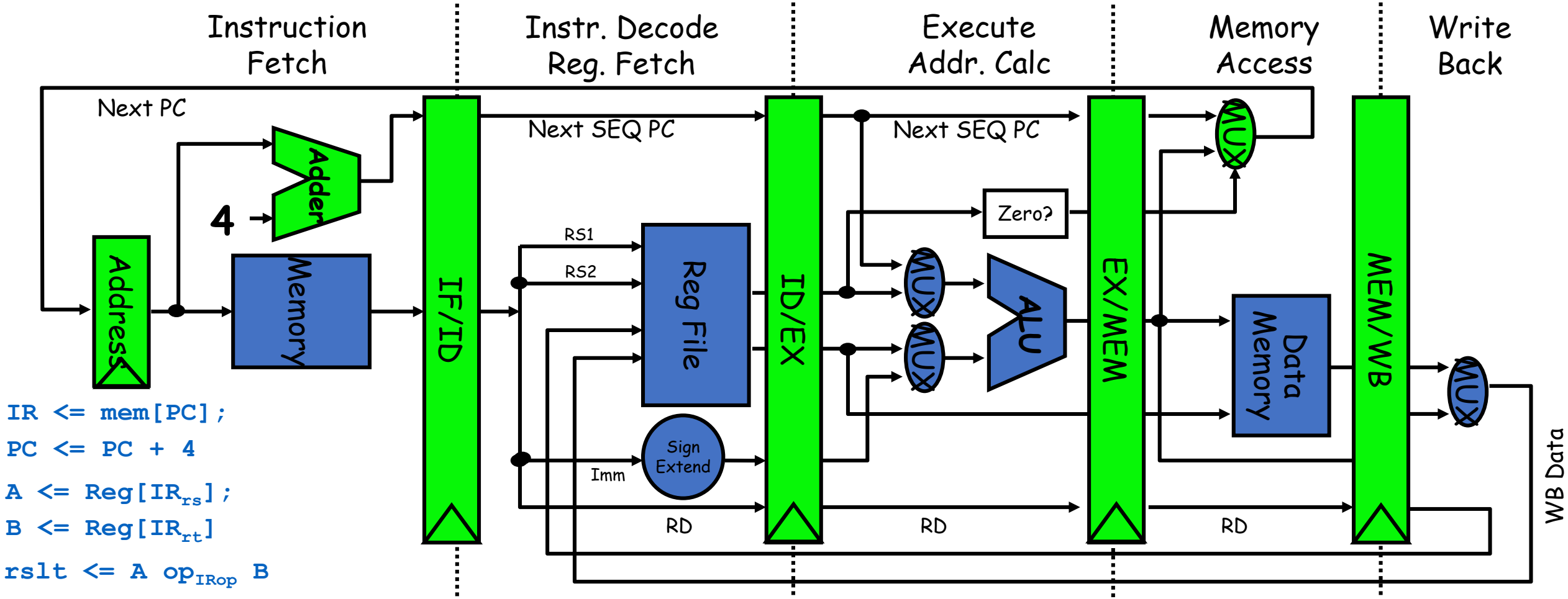
Hands-on today: Visualizing the Processor

Group based discussions from 20th August

How to read a paper and caches: 9th /13th August

Basics on Cache Attacks and PA1: 16th August

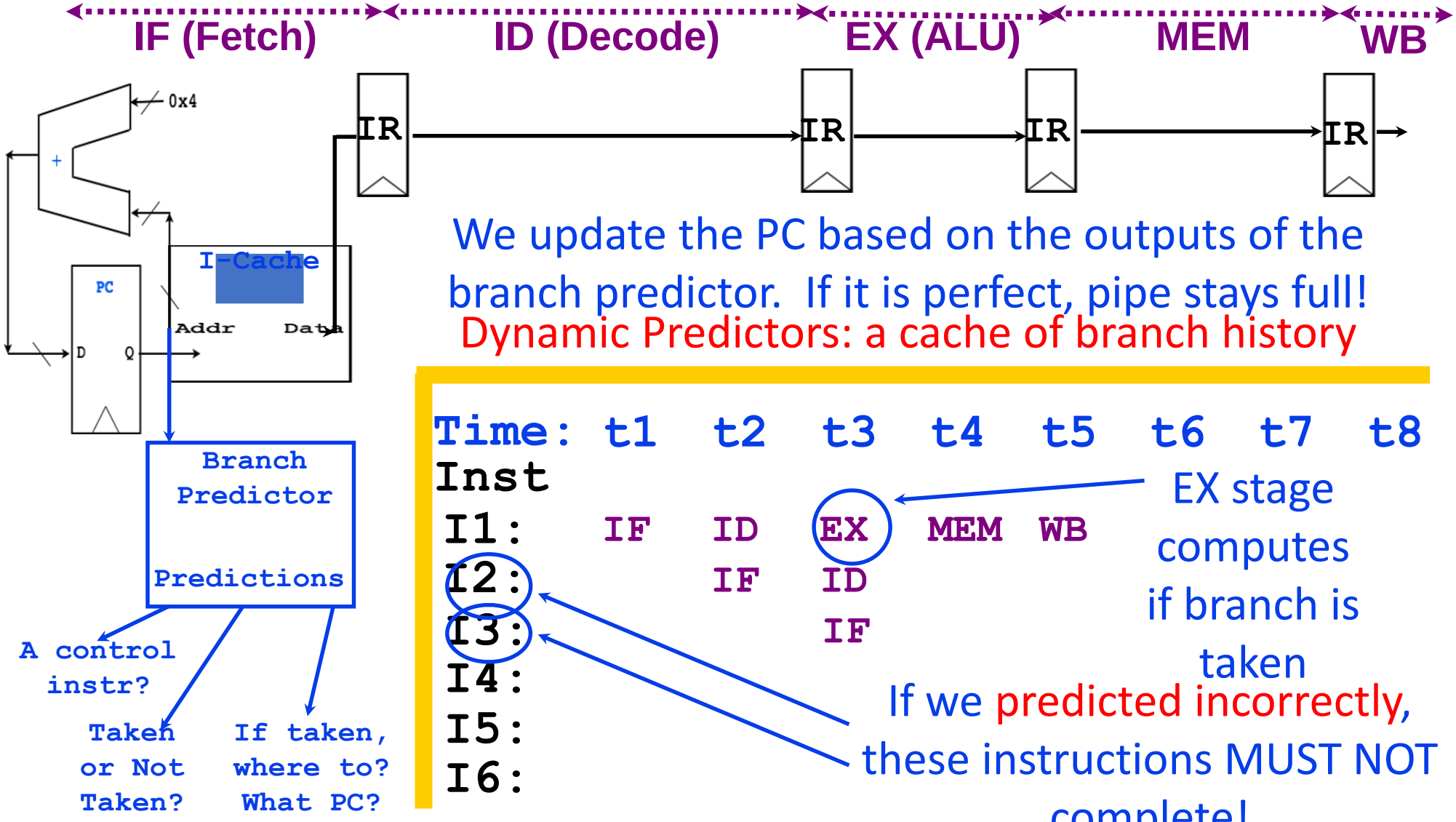
Vanilla Pipeline




```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs];
B <= Reg[IR_rt]
rslt <= A op_IRop B
WB <= rslt
Reg[IR_rd] <= WB
```

- Data stationary control
  - local decode for each instruction phase / pipeline stage

Branch Prediction and Speculative Execution



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 Target Buffer

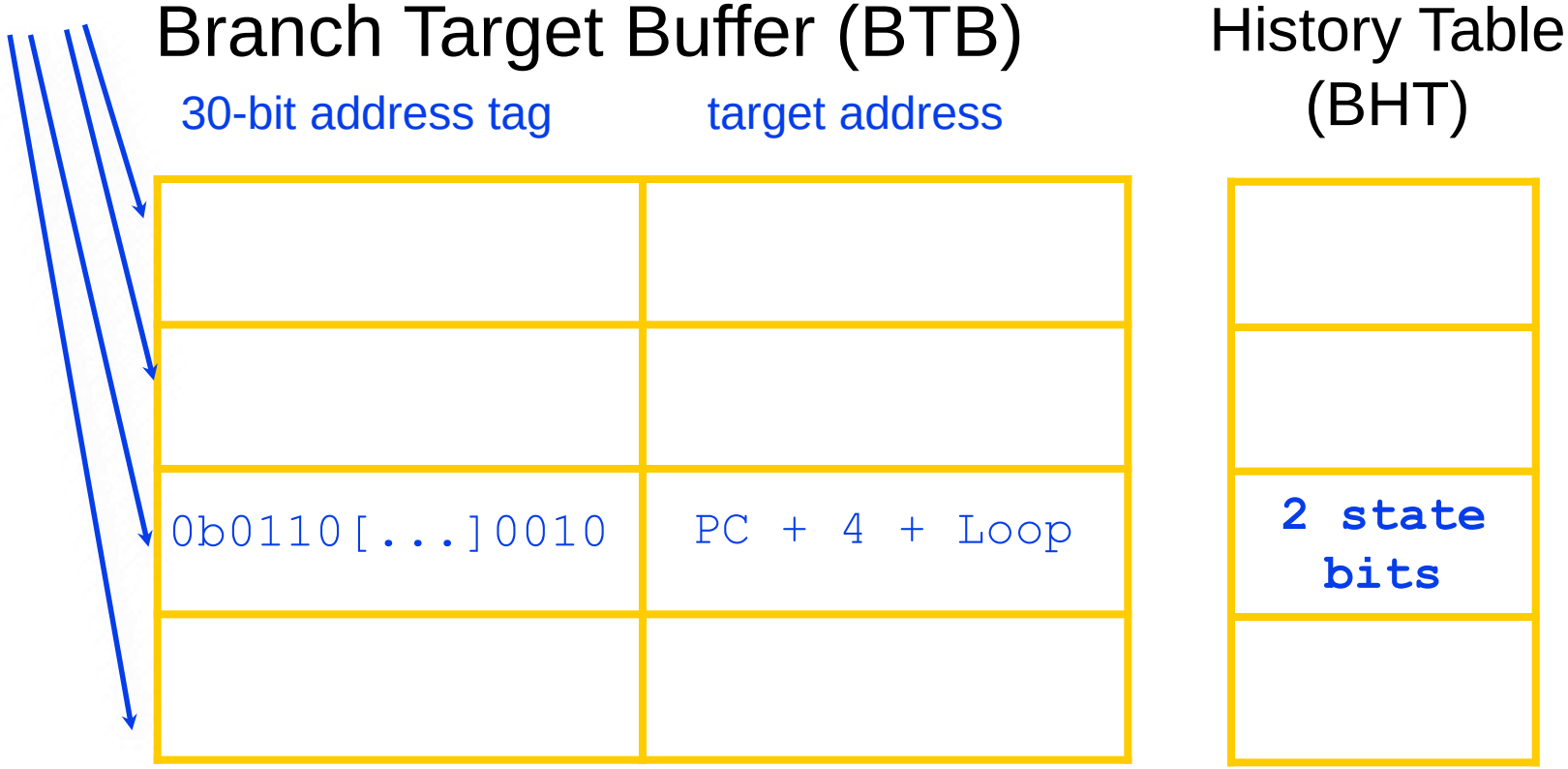
Address of branch instruction

`0b0110[...]01001000`

30 bits

Branch instruction

`BNEZ R1 Loop`

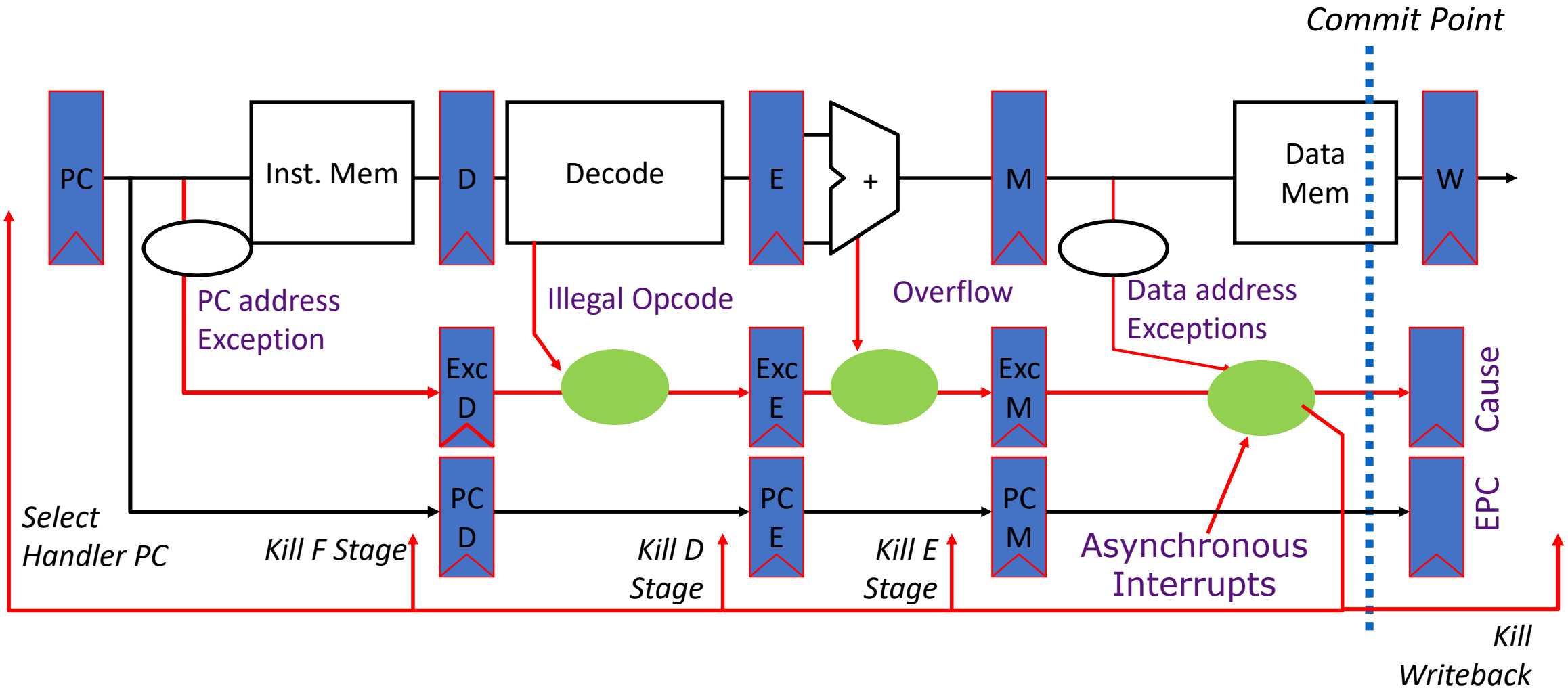


Drawn as fully associative to focus on the essentials.

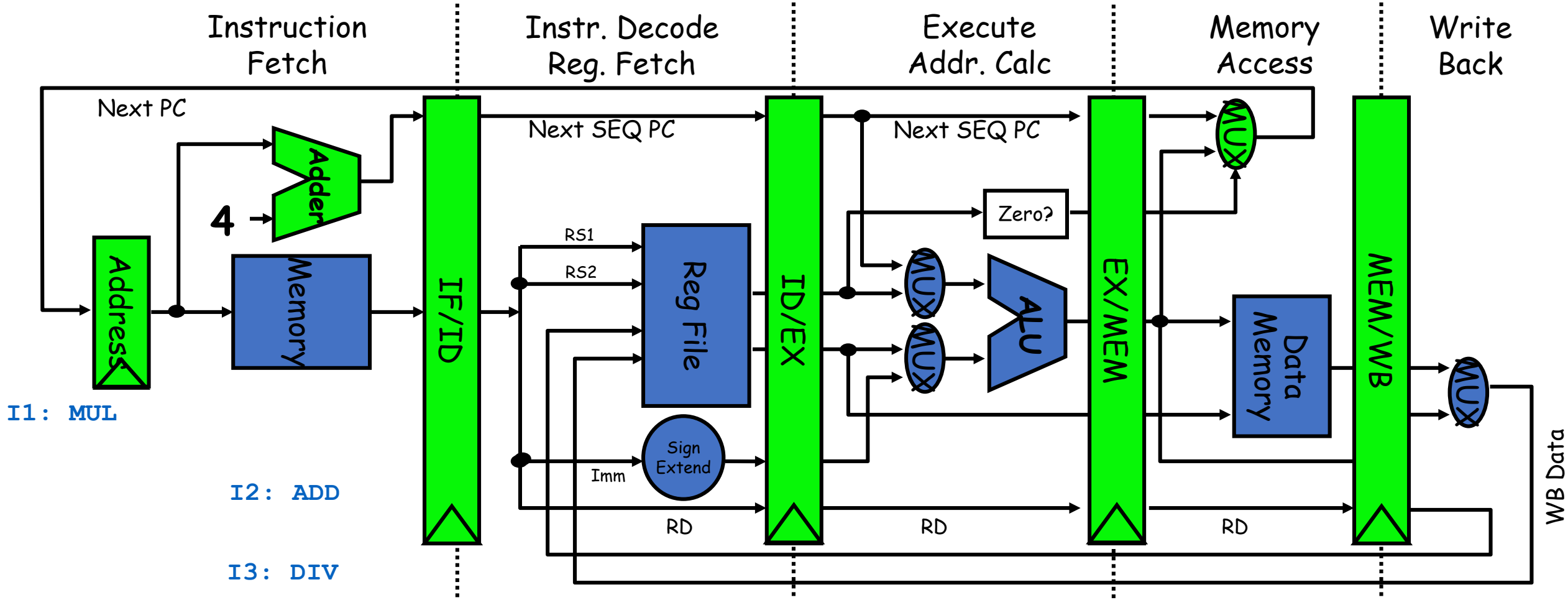
In real designs, always direct-mapped.

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

Handling Exceptions

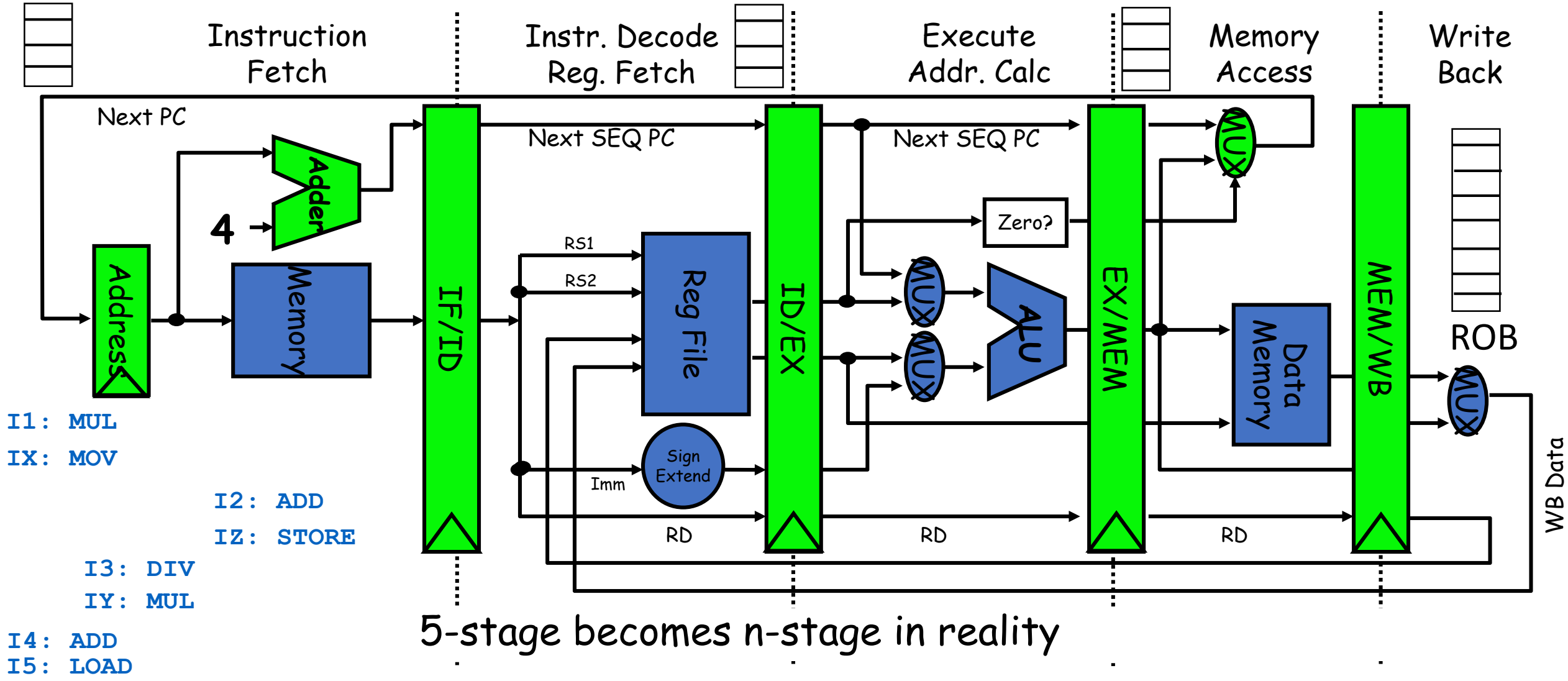


Some More: Out-of-order

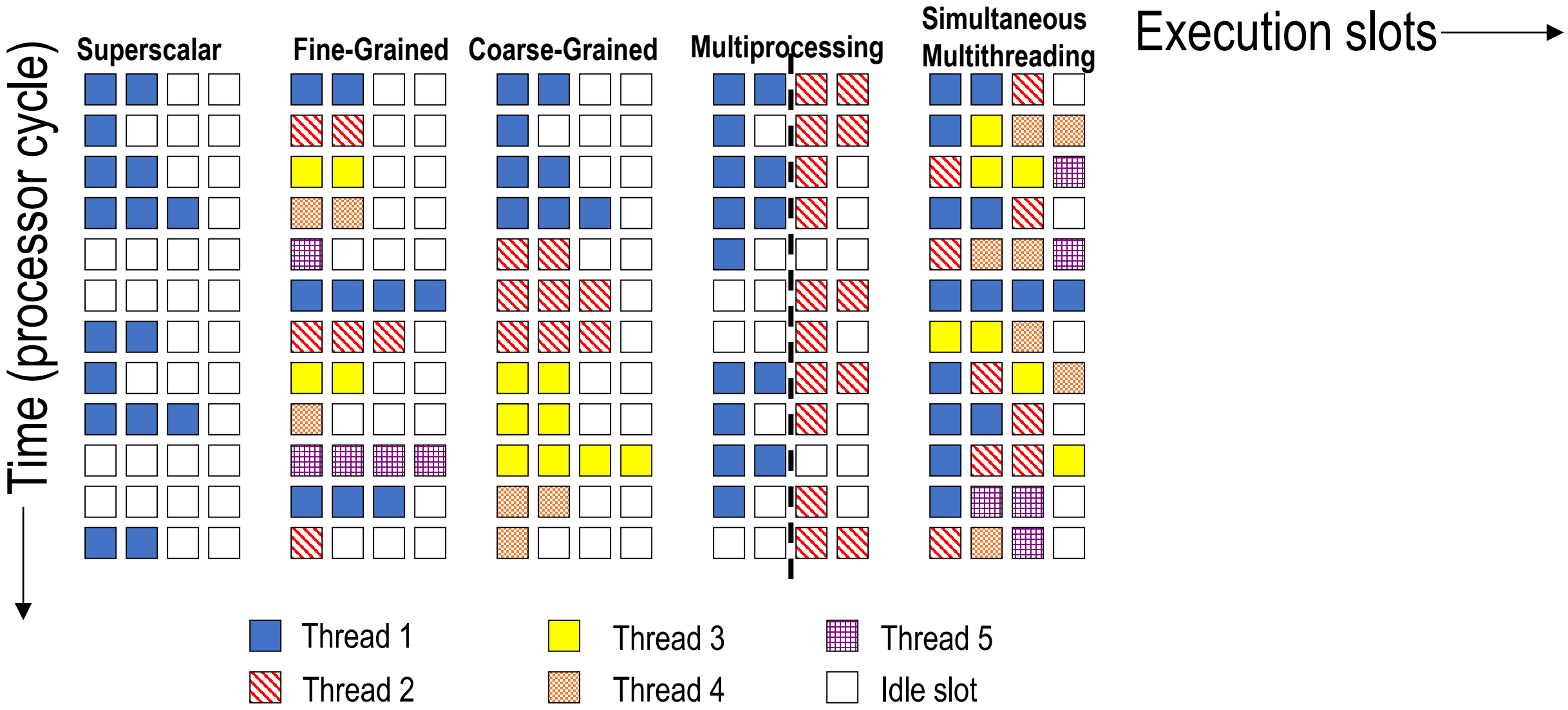


- Data stationary control
  - local decode for each instruction phase / pipeline stage

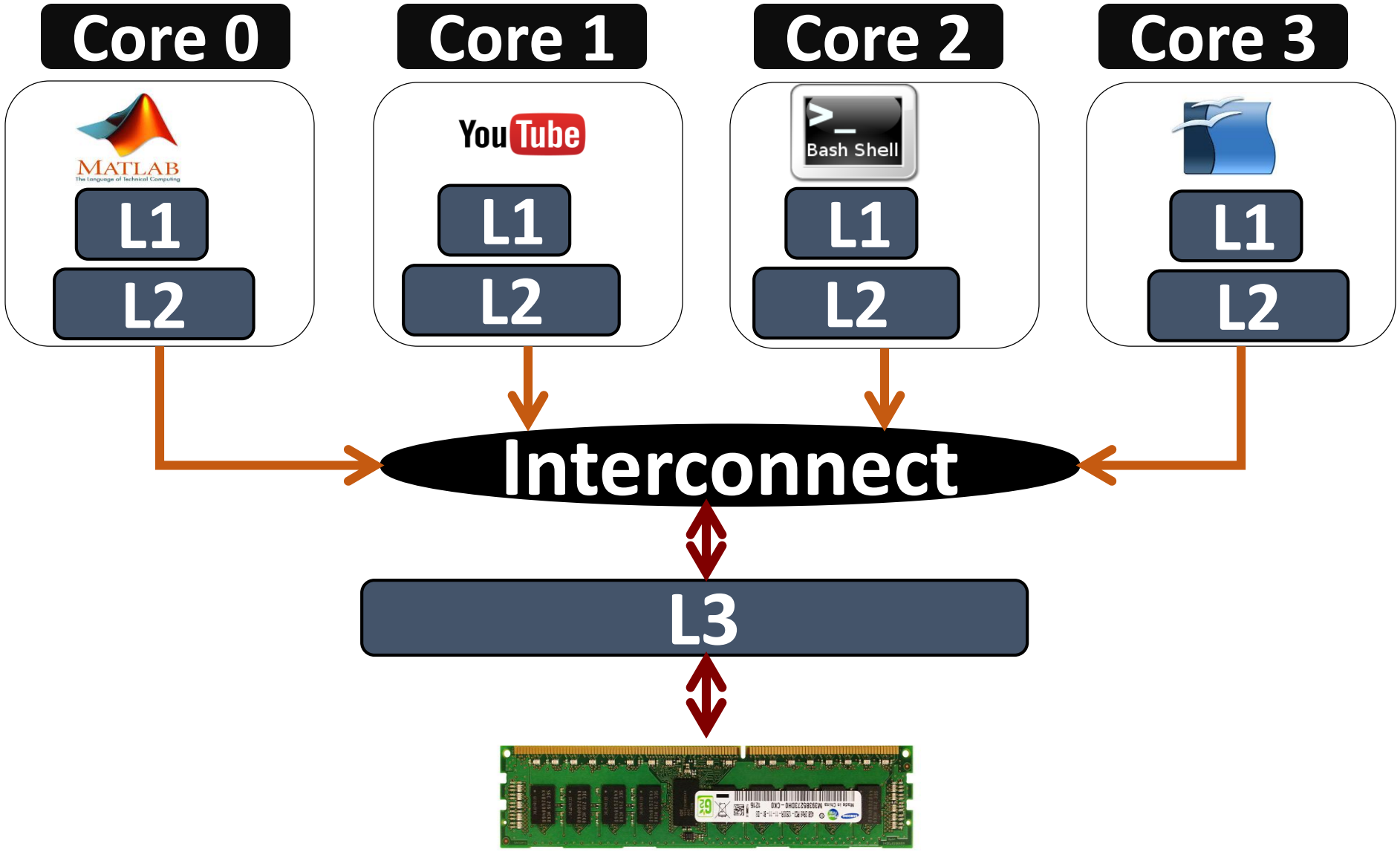
Some More: Out-of-order + Multi-Issue (a.k.a superscalar)



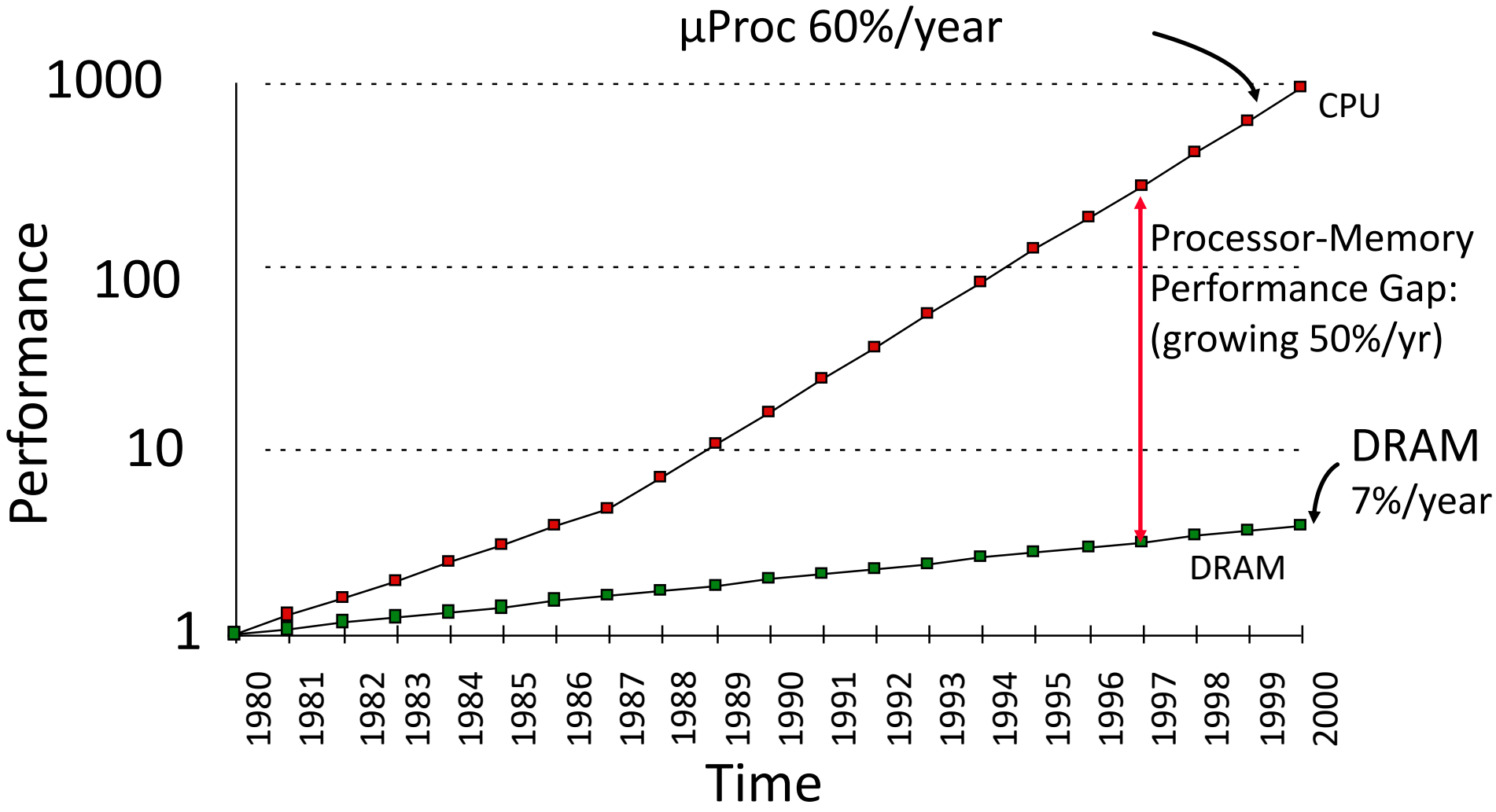
TLP: Multithreading



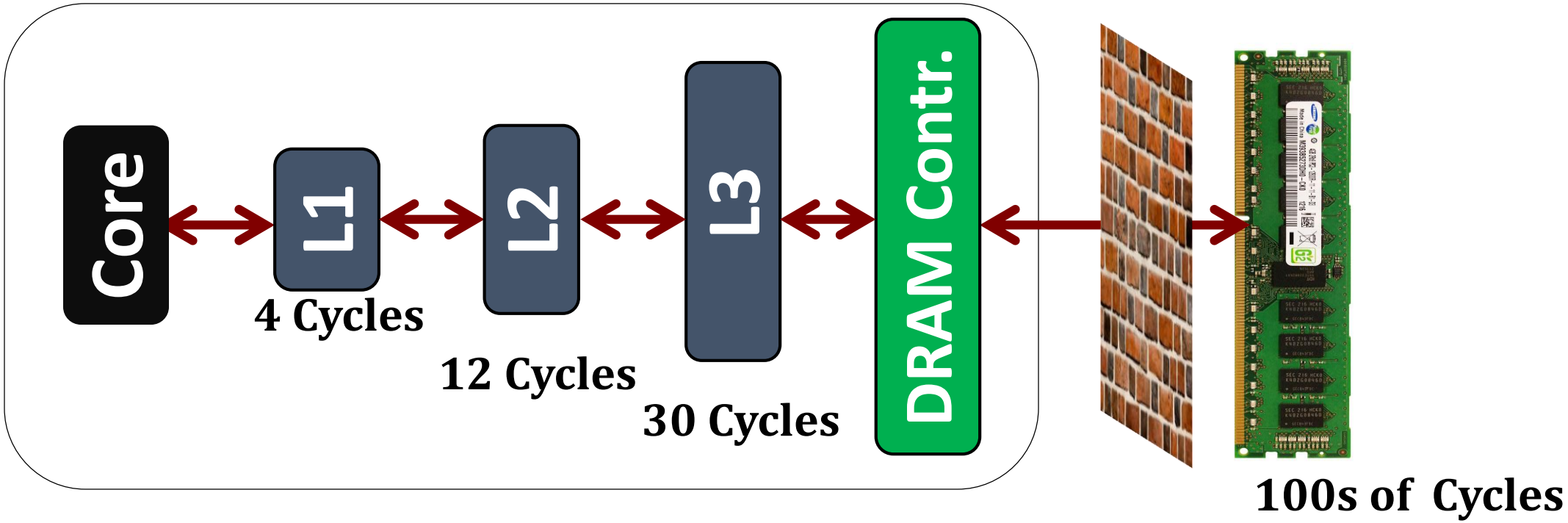
CACHES: WHERE ARE THEY



Memory Wall Problem



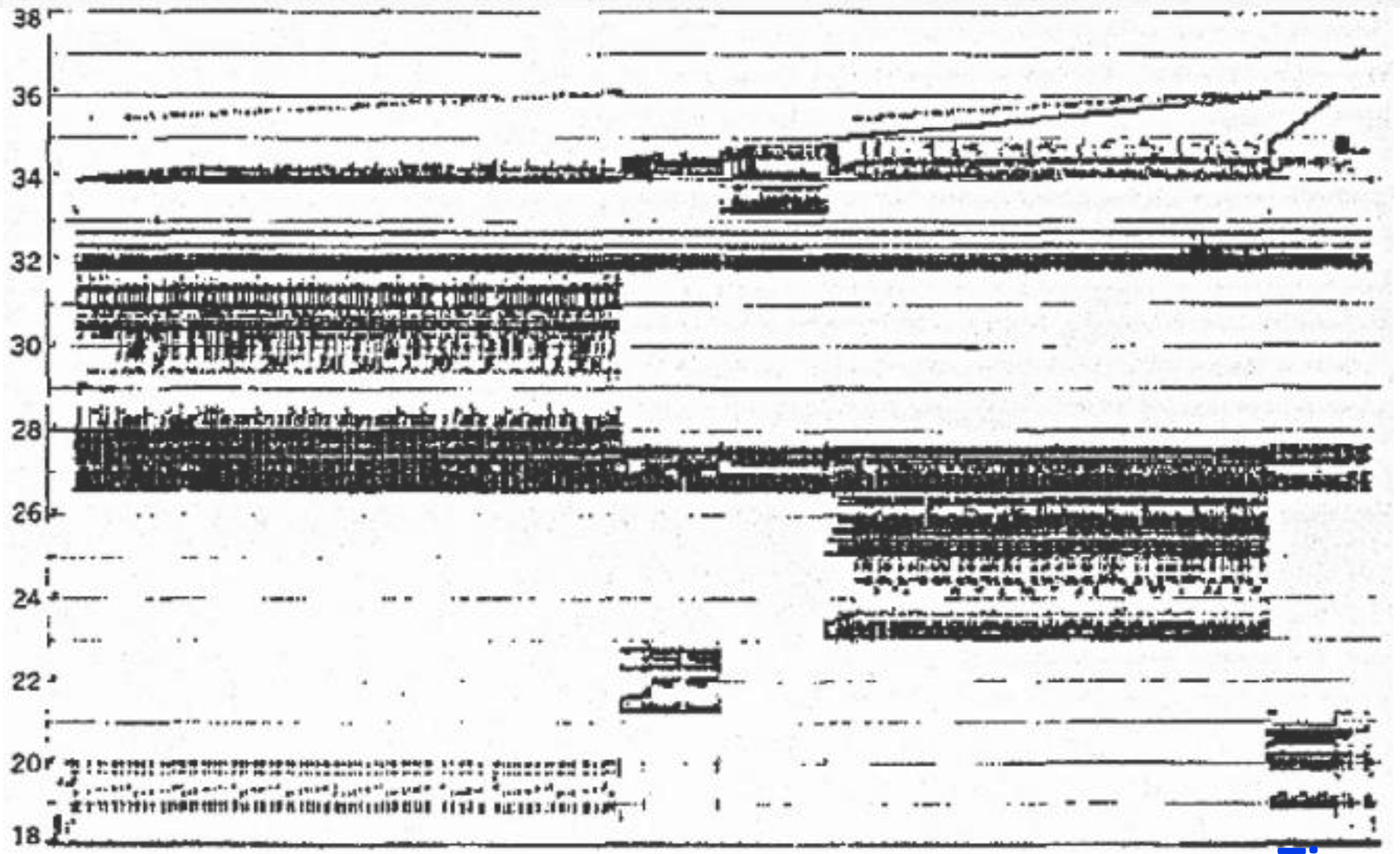
WHY ARE THEY?



Latency wall

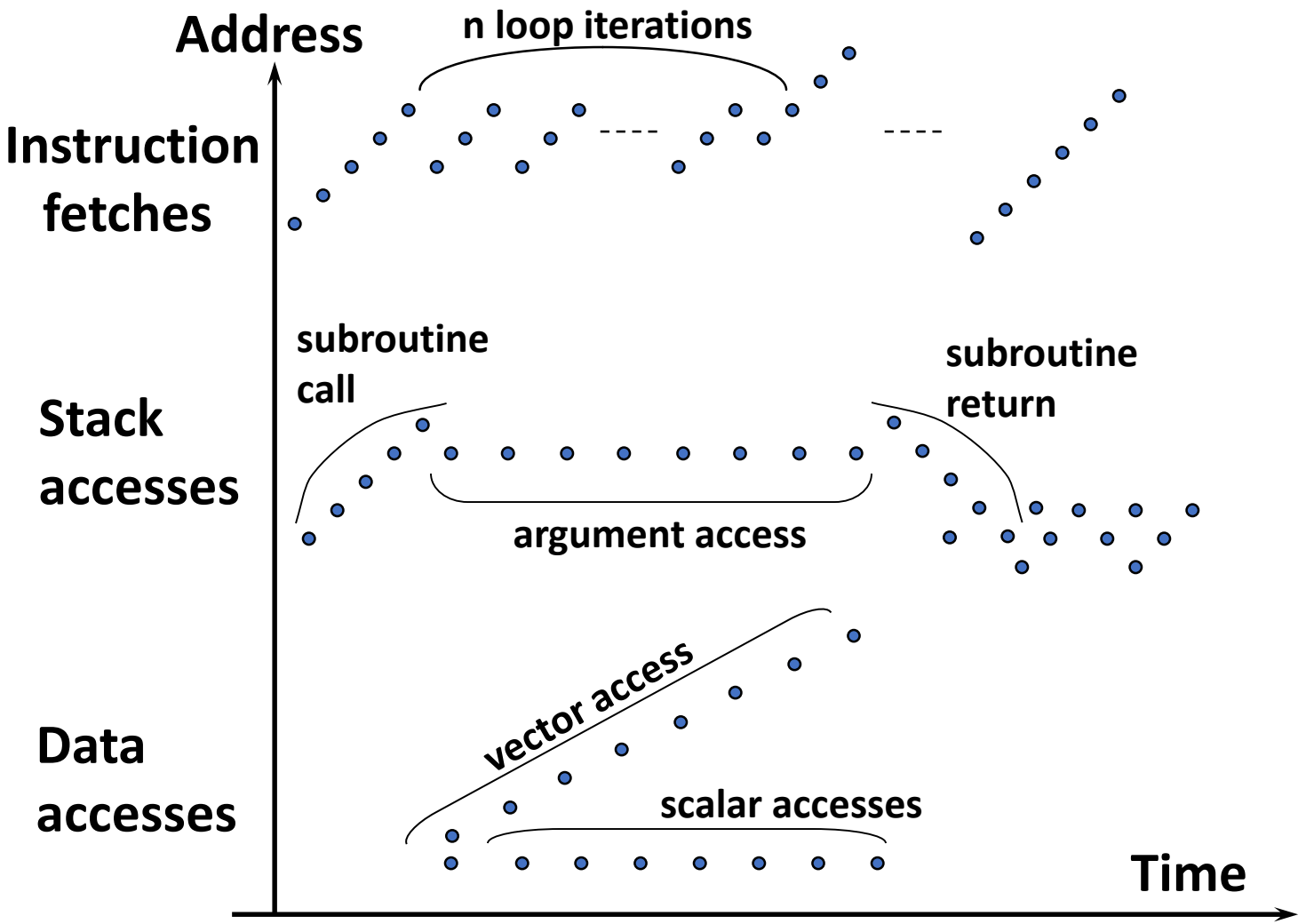
Bandwidth wall

Access Patterns



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

Examples

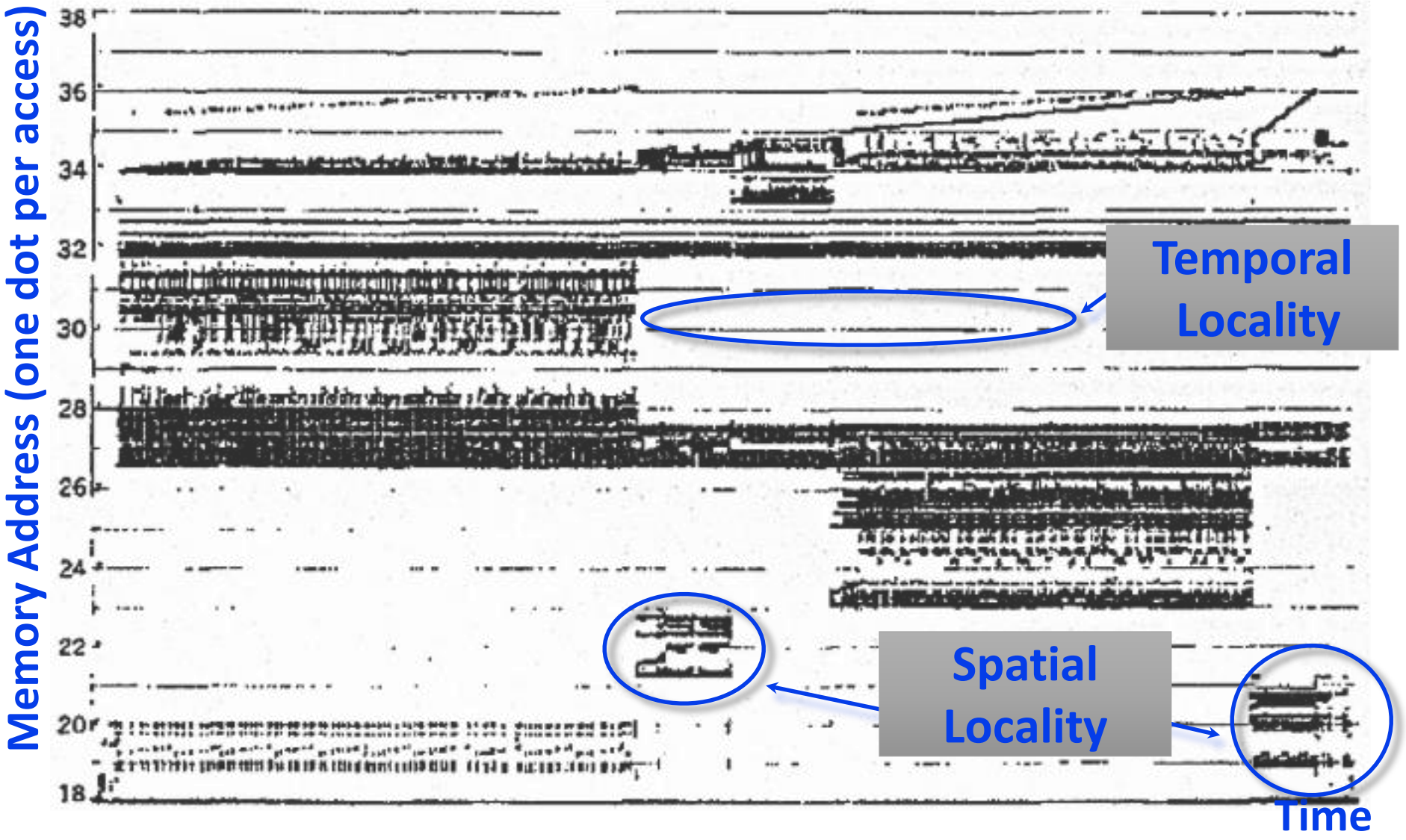

Address
n loop iterations
Instruction fetches
subroutine call
subroutine return
Stack accesses
argument access
vector access
scalar accesses
Data accesses
Time

Locality of Reference

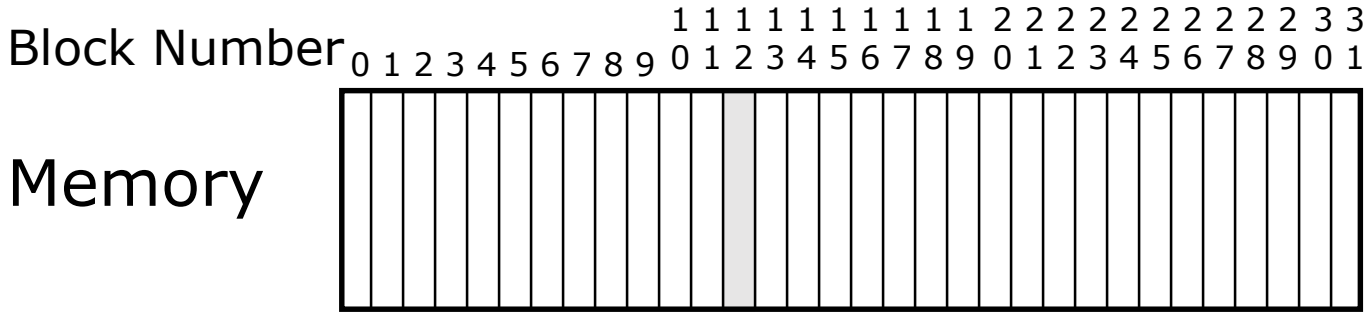
- **Temporal Locality**: If a location is referenced it is likely to be referenced again in the near future.

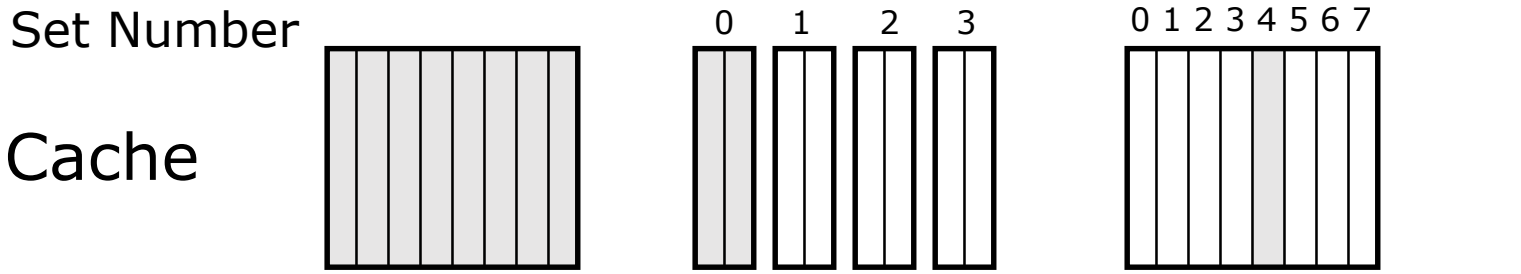
- **Spatial Locality**: If a location is referenced it is likely that locations near it will be referenced in the near future.

Again


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

Placement Policy





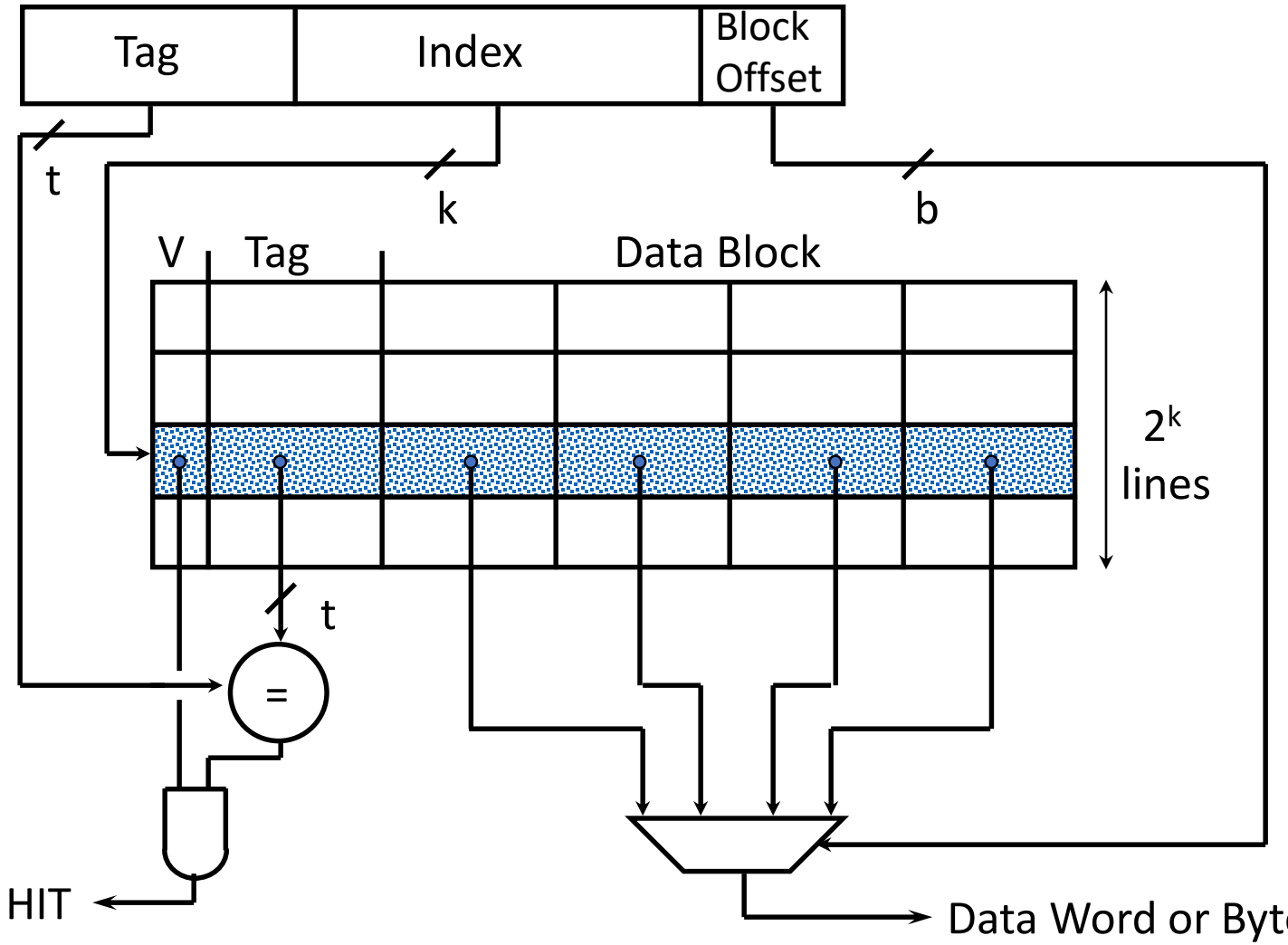
Fully Associative
anywhere

(2-way) Set Associative
anywhere in set 0
(12 mod 4)

Direct Mapped
only into block 4
(12 mod 8)

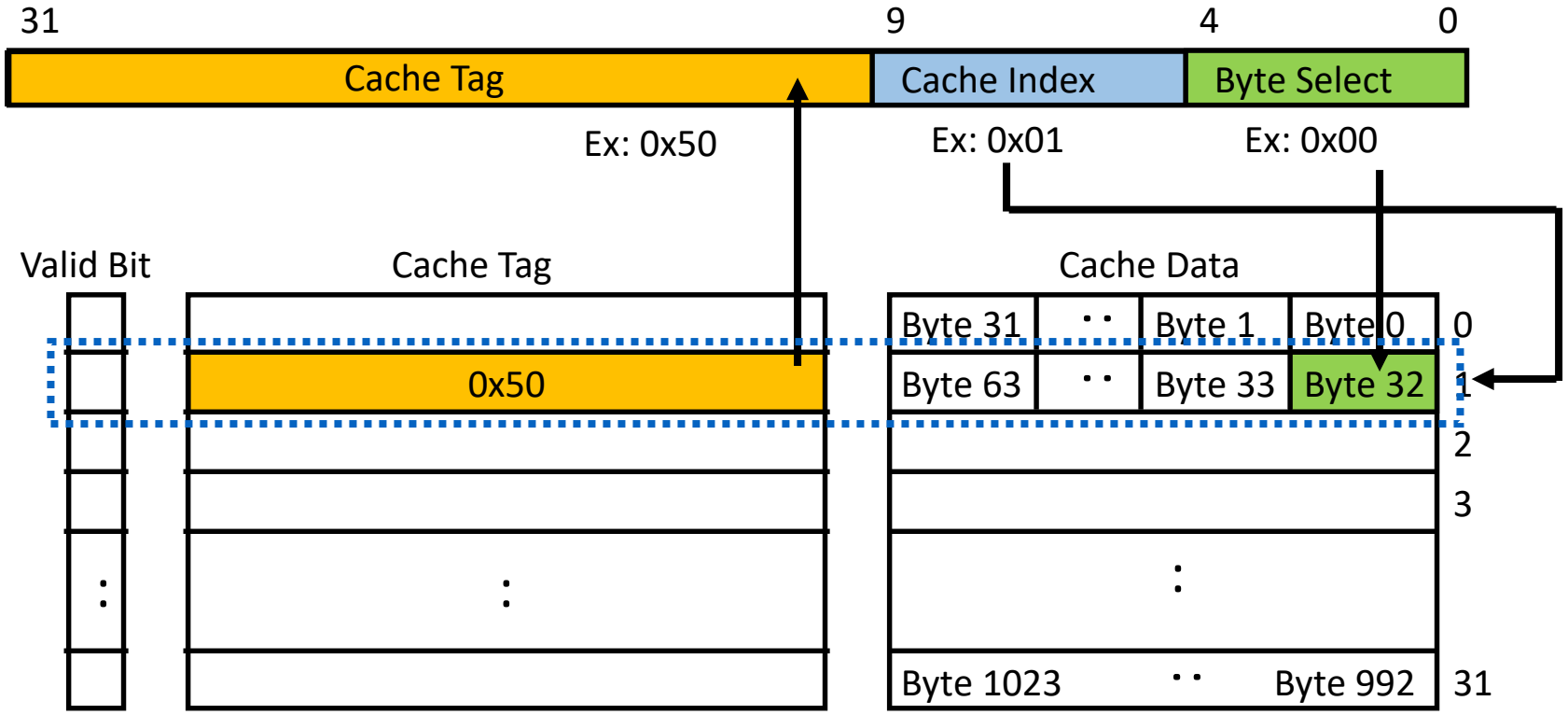
block 12 can be placed

Direct Mapped

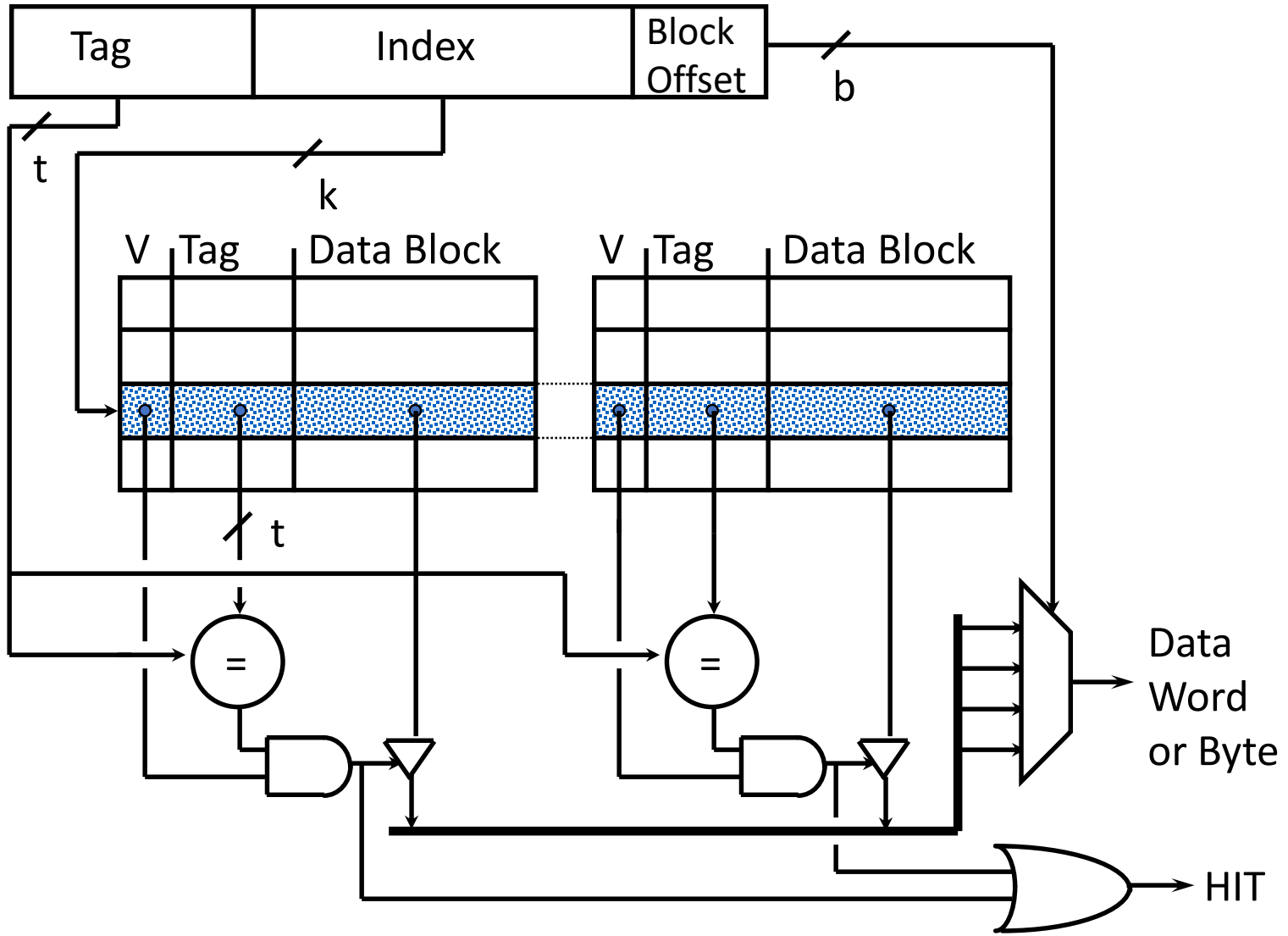


In reality, tag-store is placed separately

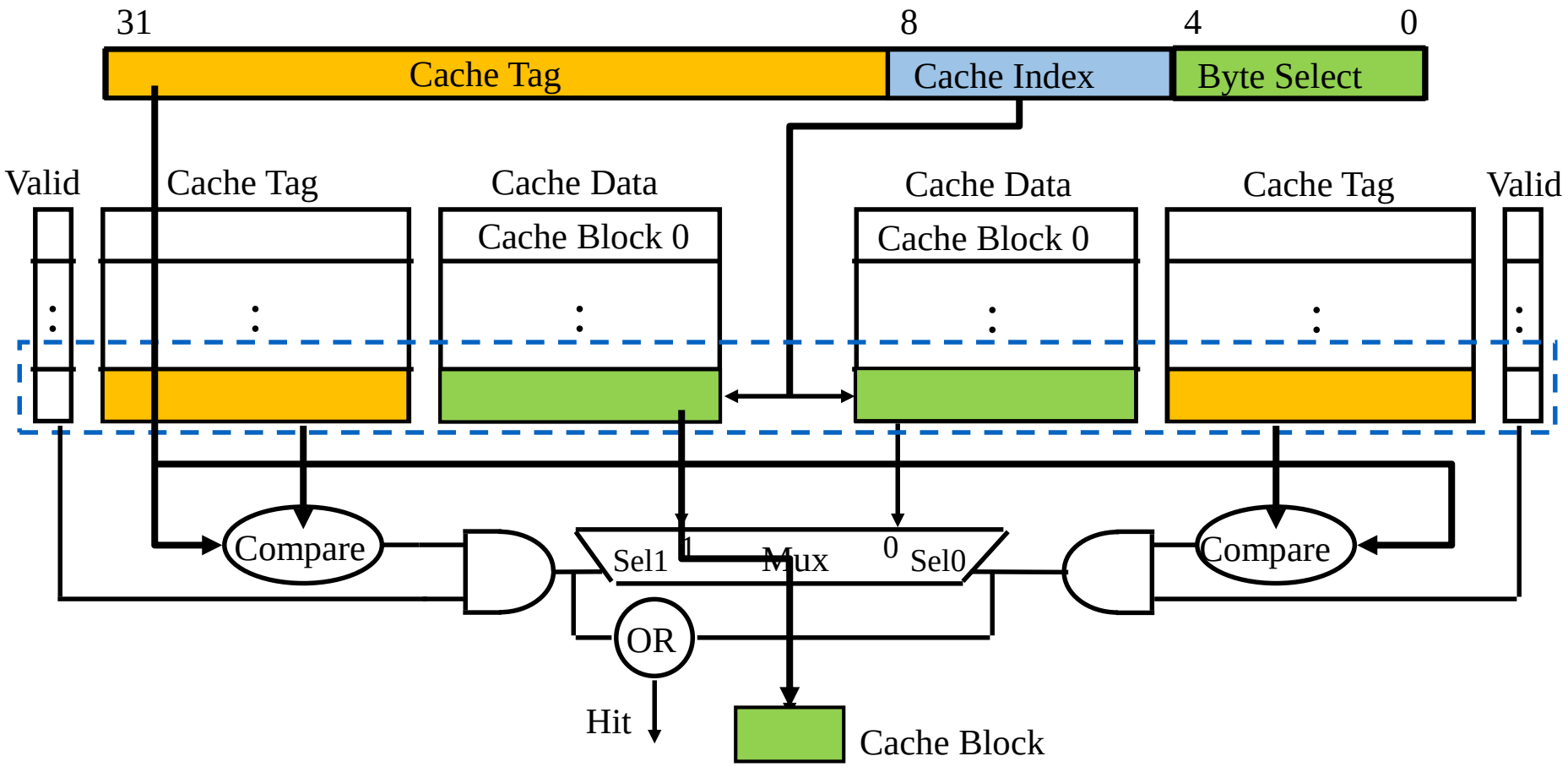
An Example


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
0
Byte 63
Byte 33
Byte 32
1
2
3
Byte 1023
Byte 992
31

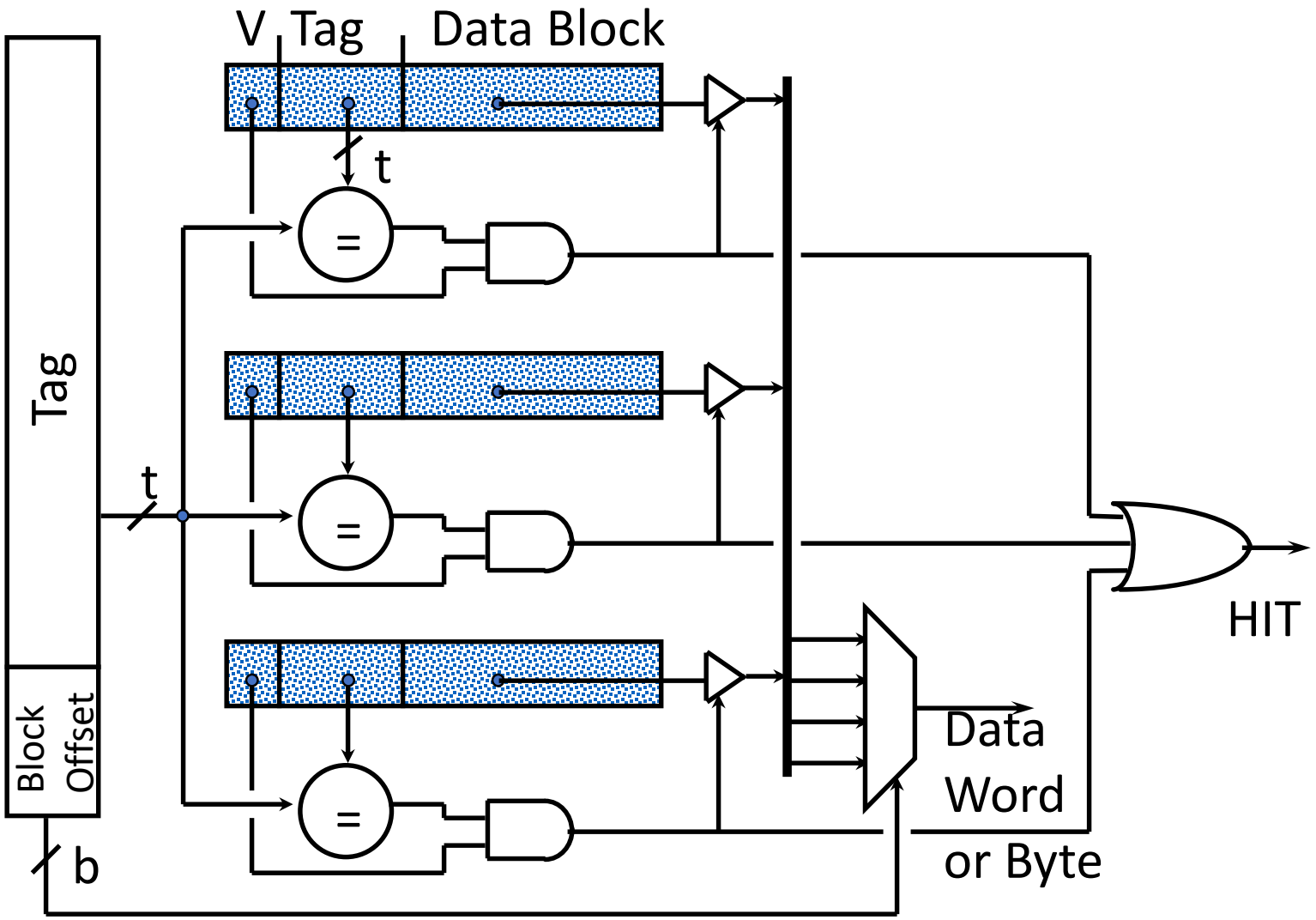
Set-Associative



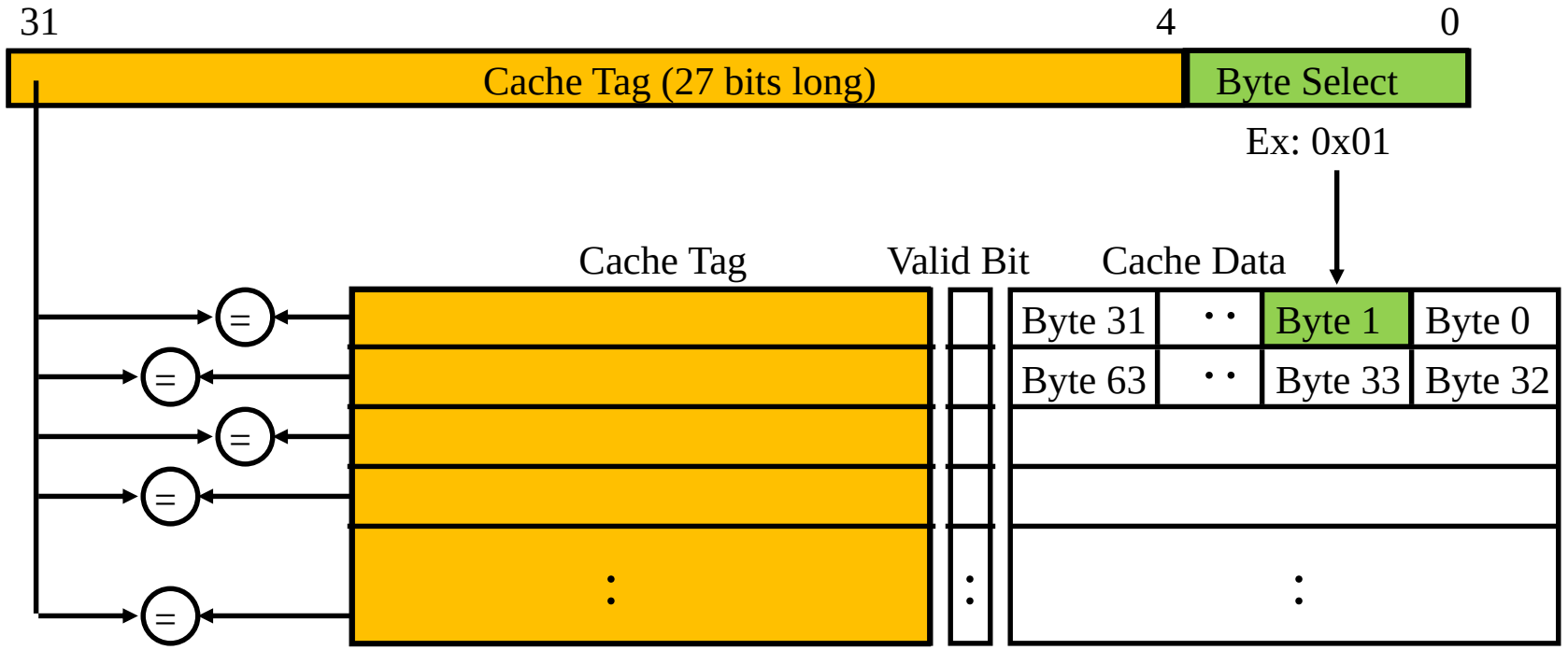
An Example


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



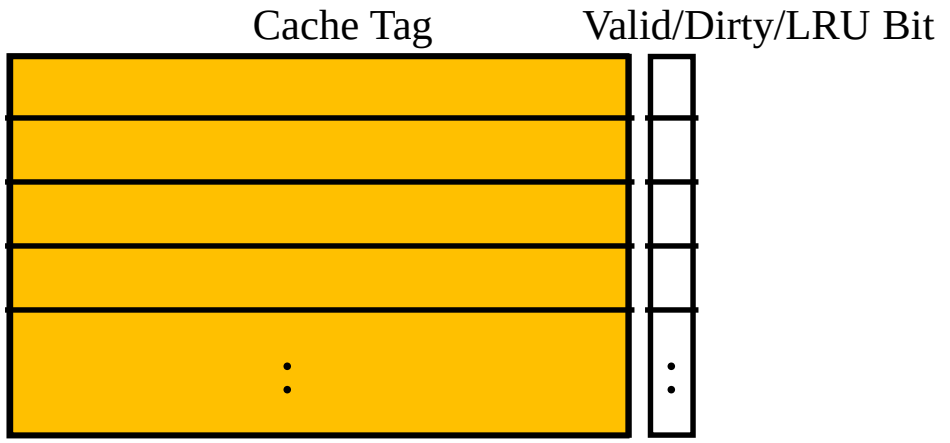
An Example

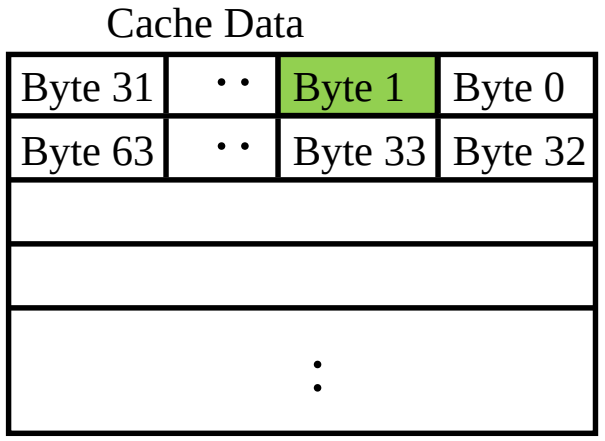

31
4
0
Cache Tag (27 bits long)
Byte Select
Ex: 0x01
Cache Tag
Valid Bit
Cache Data
Byte 31
Byte 1
Byte 0
Byte 63
Byte 33
Byte 32
=
=
=
=
=

What's in Tag Store?

- Valid bit
- Tag
- Replacement policy bits

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





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

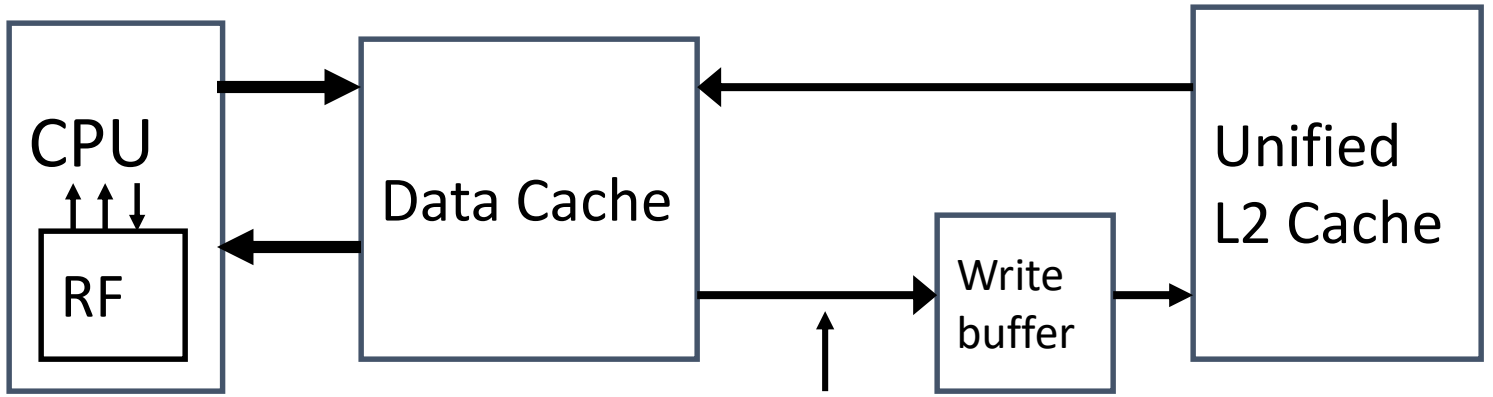
Reads are not Writes

- 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

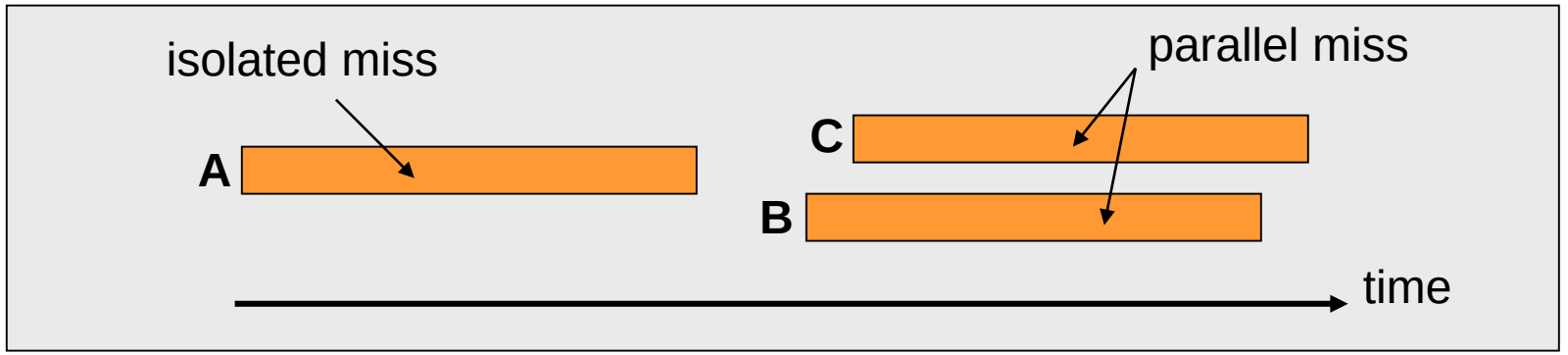
To improve performance:

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

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?

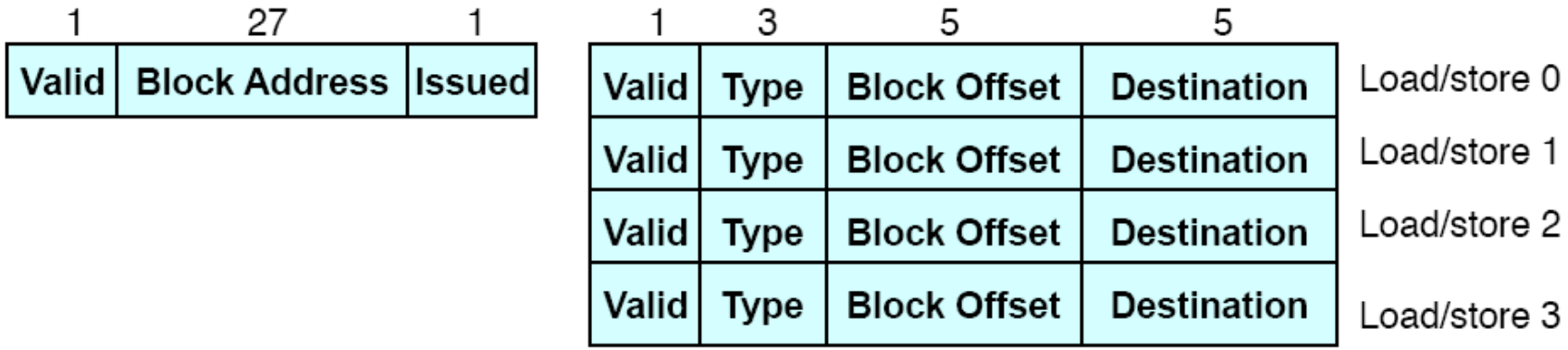


    - Enables latency tolerance: overlaps latency of different misses
  - How to generate multiple misses?
    - Out-of-order execution, multithreading, prefetching

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
Load/store 1
Load/store 2
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*

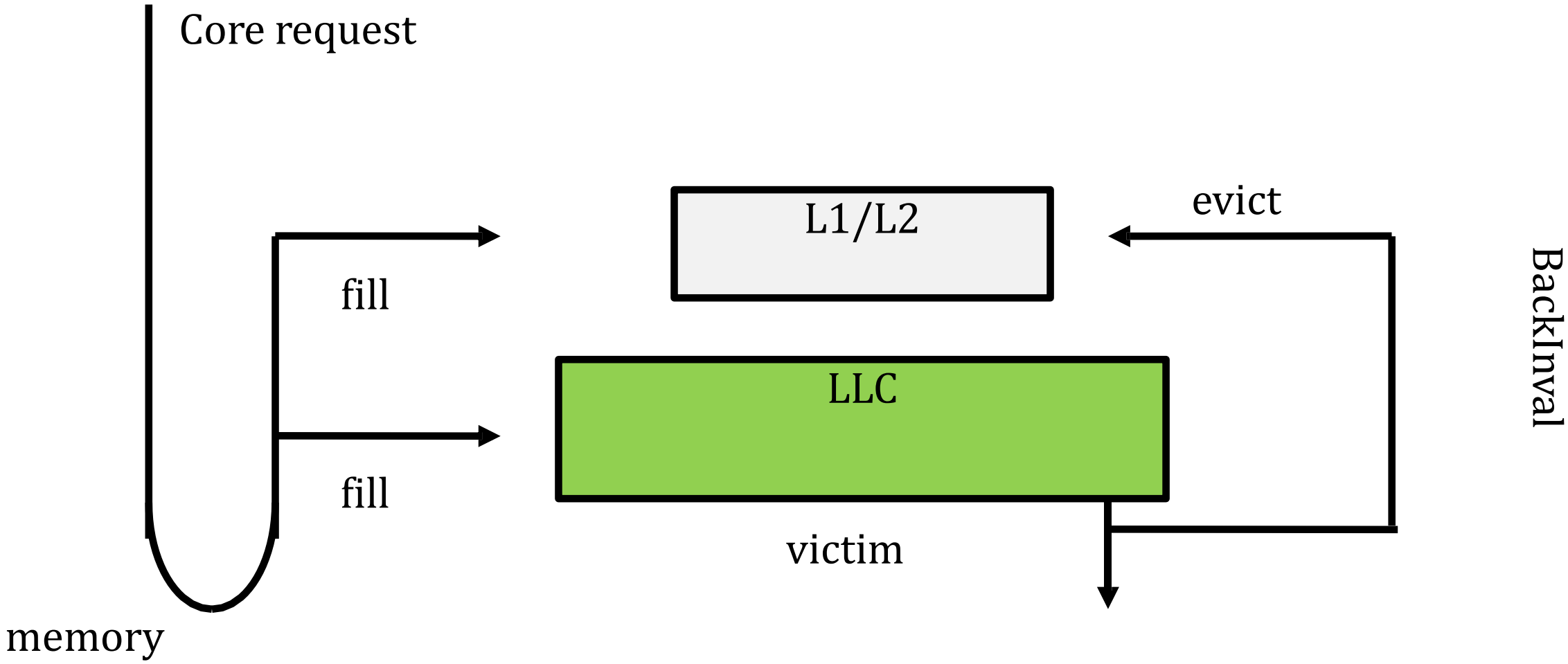
Cache Knobs and Performance

- Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

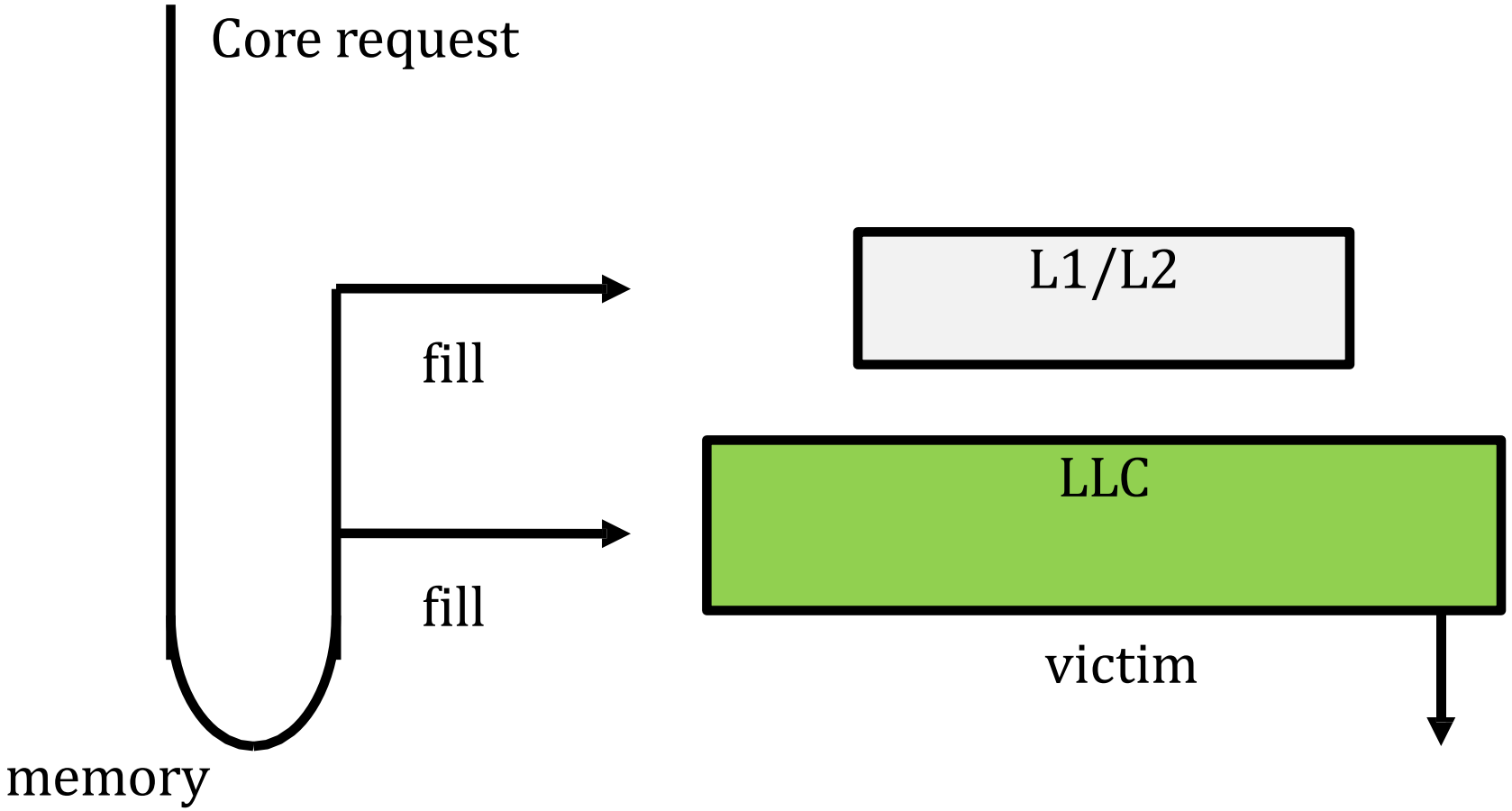
- Higher associativity
  + reduces conflict misses
  - may increase hit time

- Larger line size
  + reduces compulsory and capacity (reload) misses
  - increases conflict misses and miss penalty

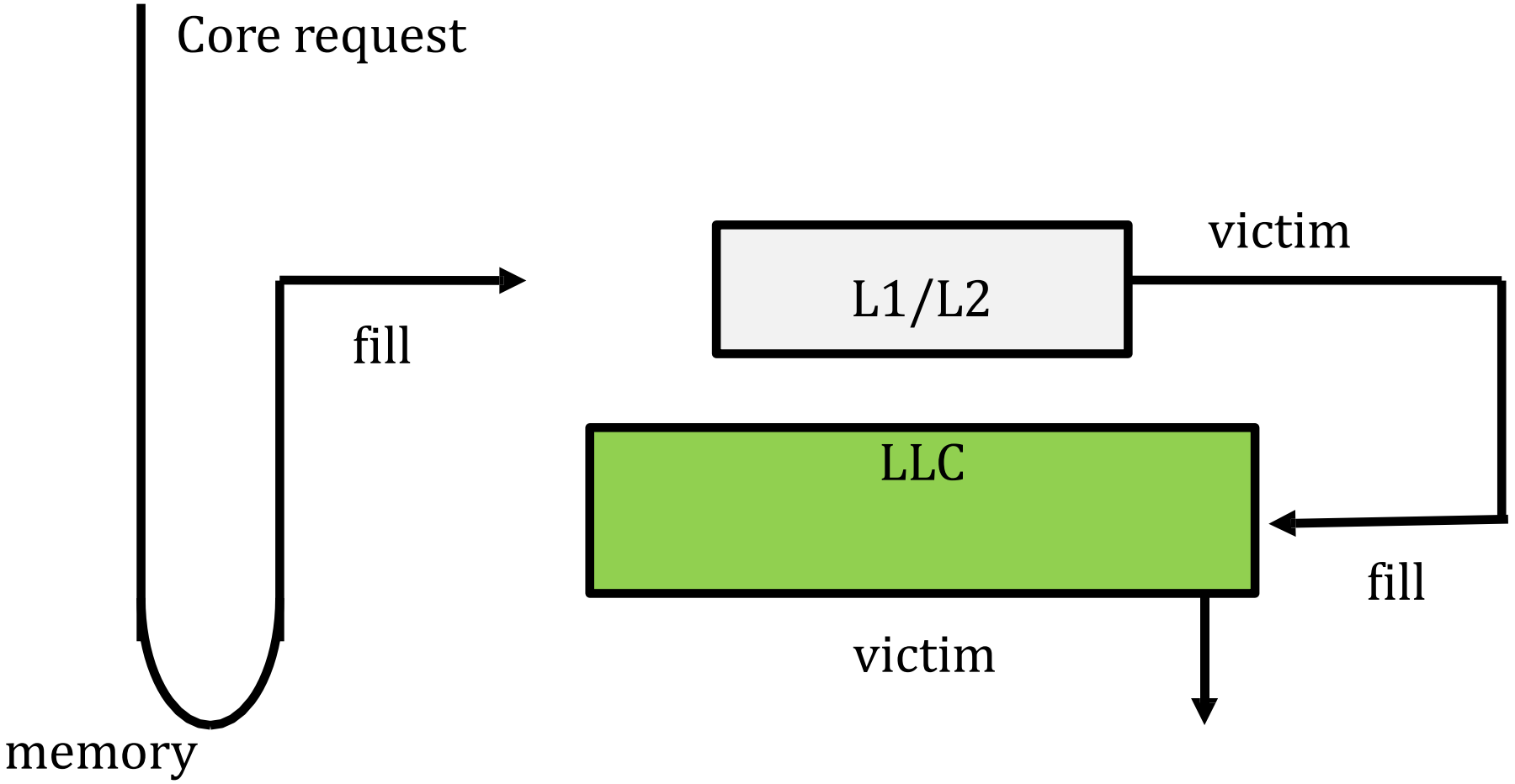
Cache Hierarchy: Inclusive



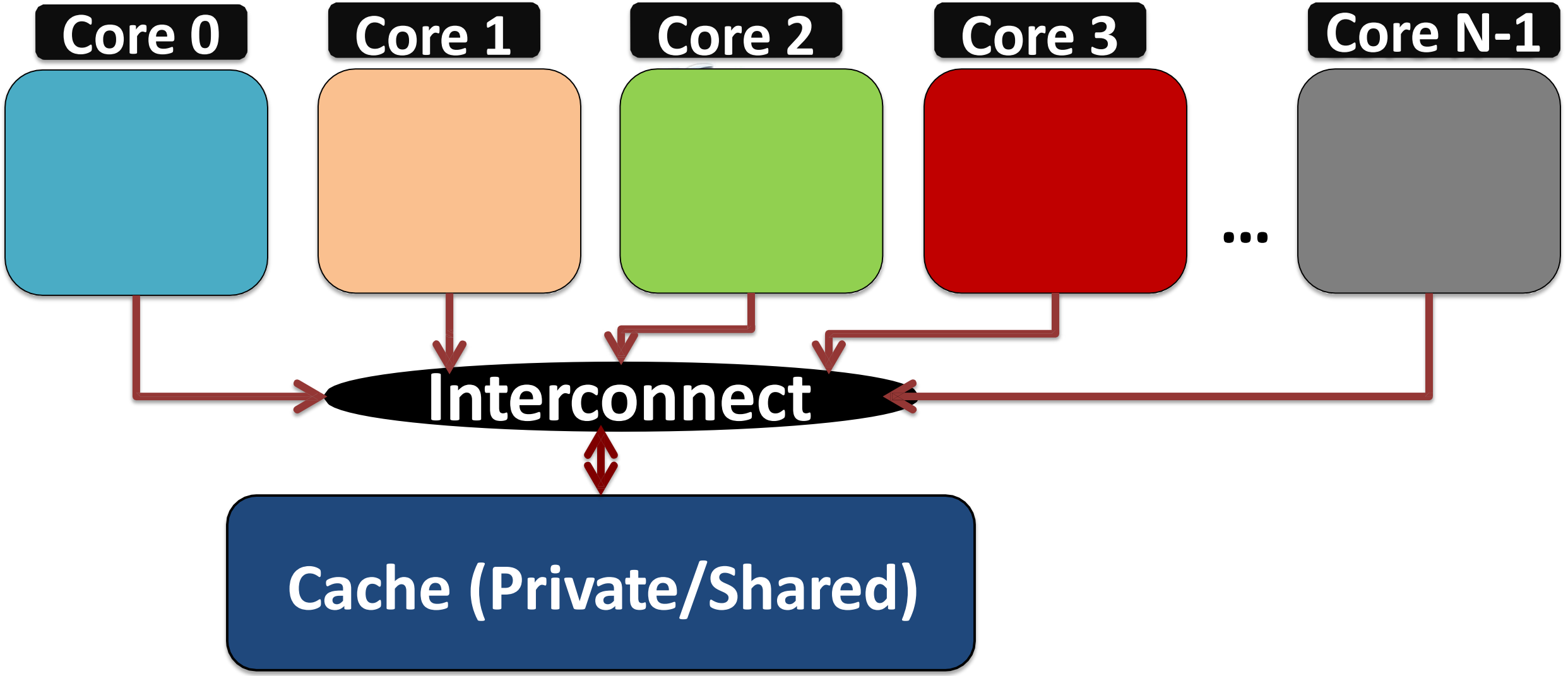
Non-inclusive



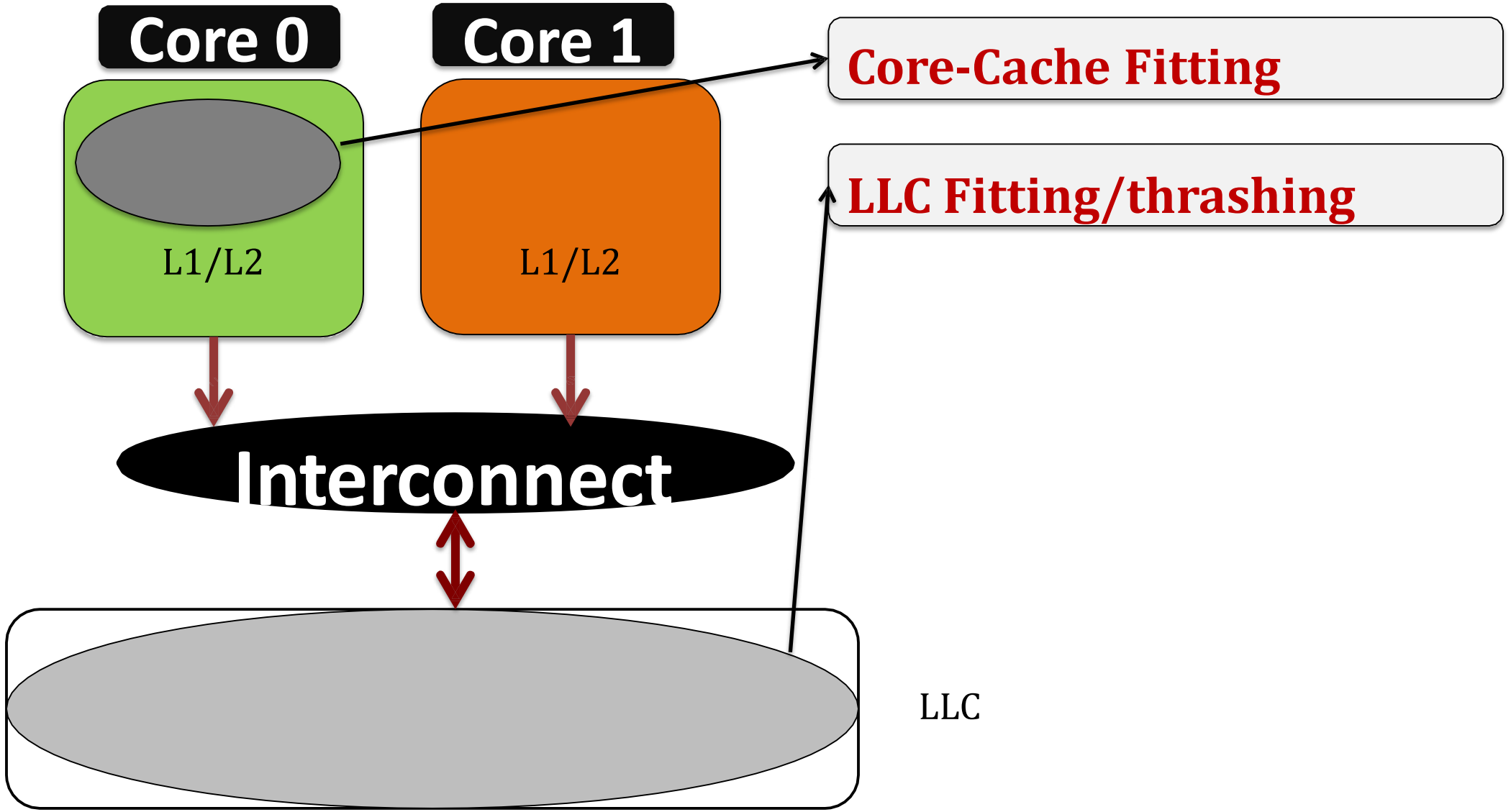
Exclusive



LLC: Shared or Private?



Application Behavior



Shared LLC

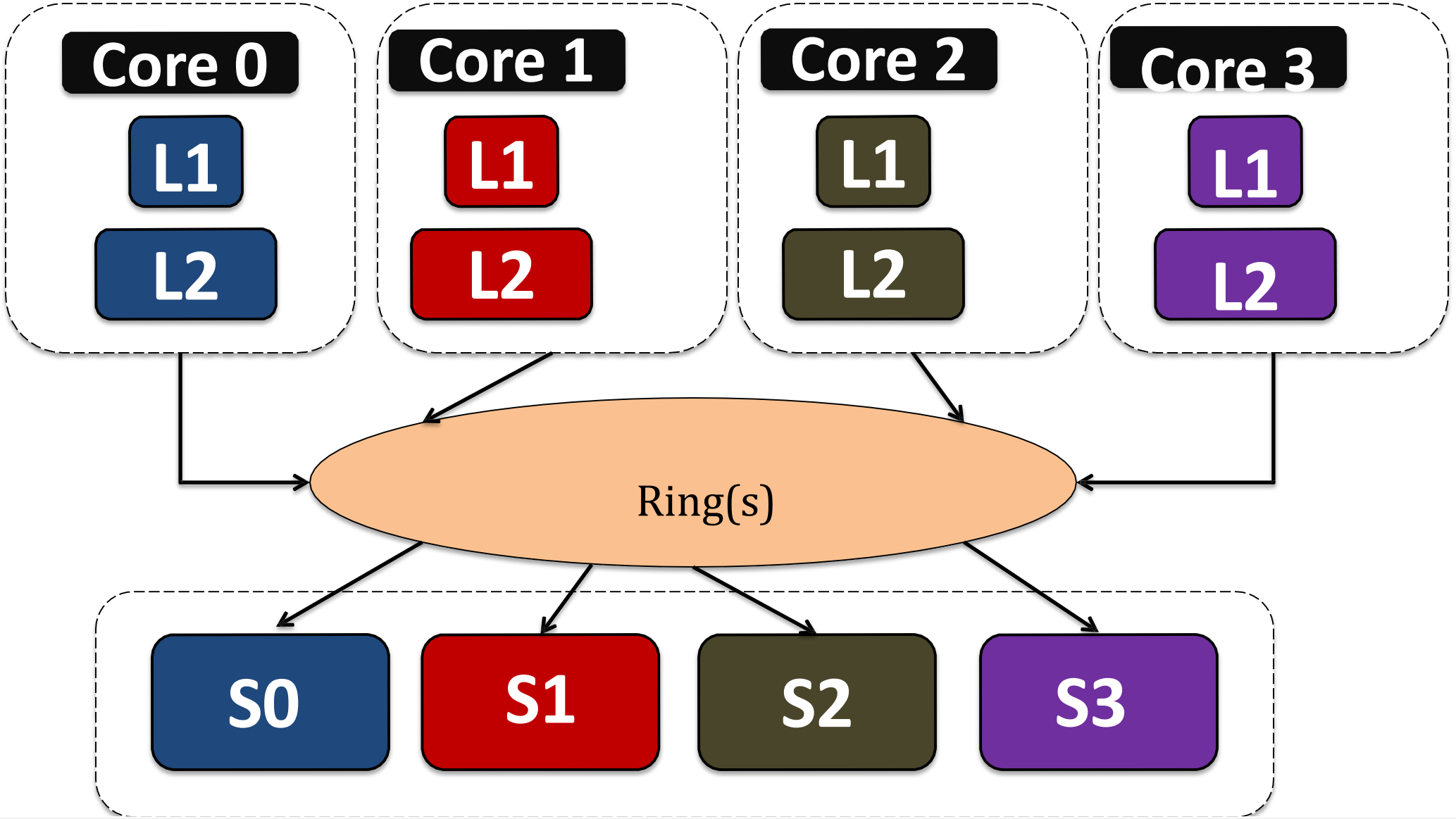
Shared LLC provides a good tradeoff for all kinds of apps.

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

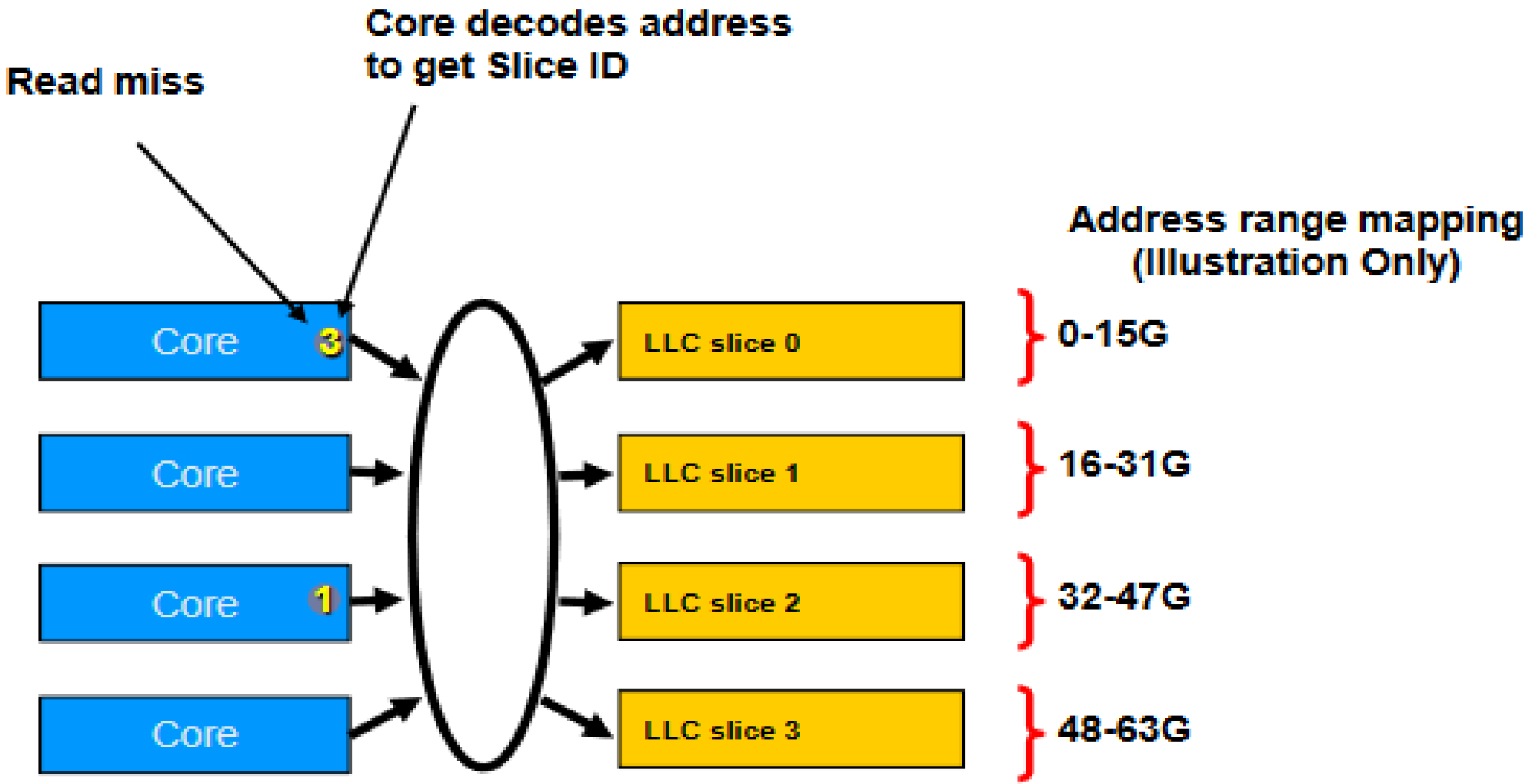
However bandwidth is an issue ☹

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?

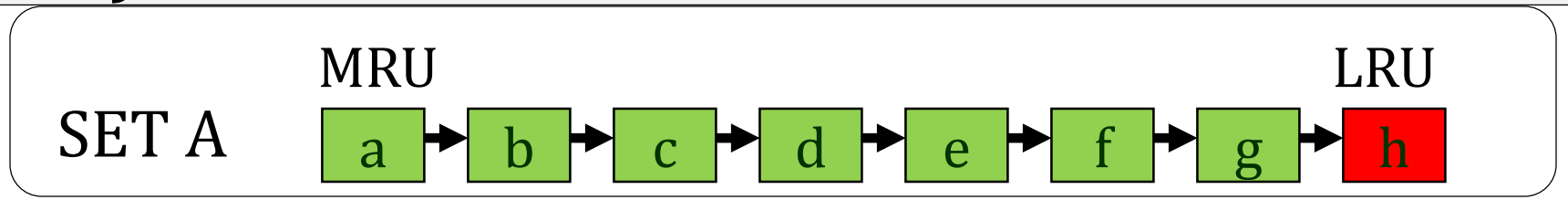
Belady

- 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?

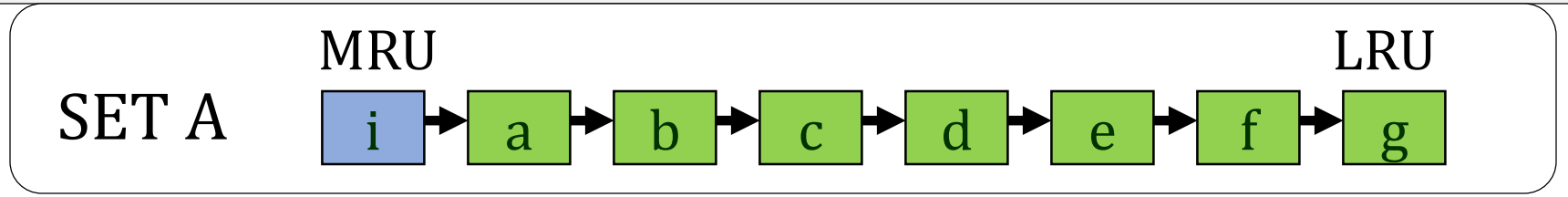
- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

LRU - 101

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



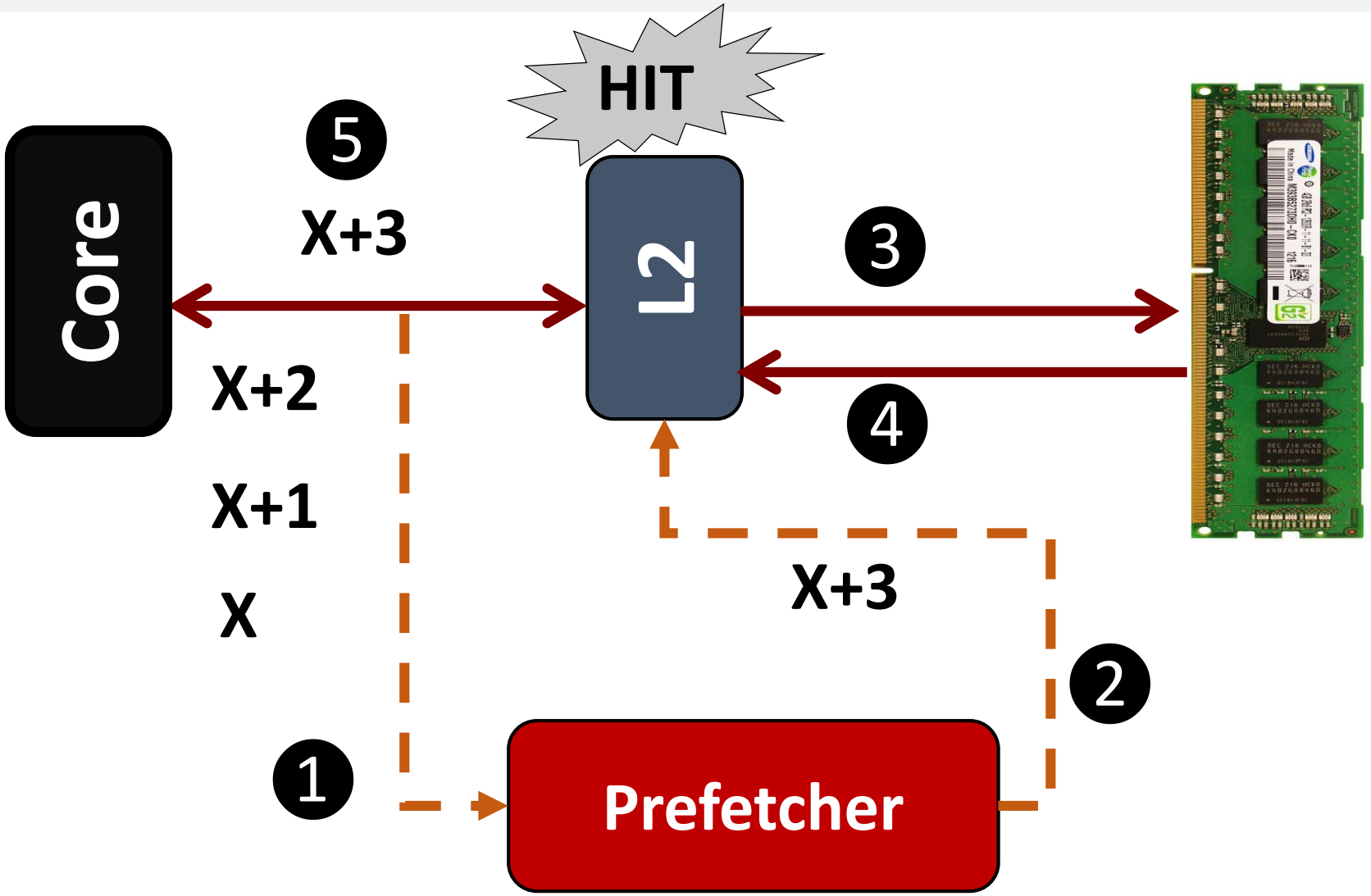
Cache Insertion Policy: New block *i* inserted into MRU.



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

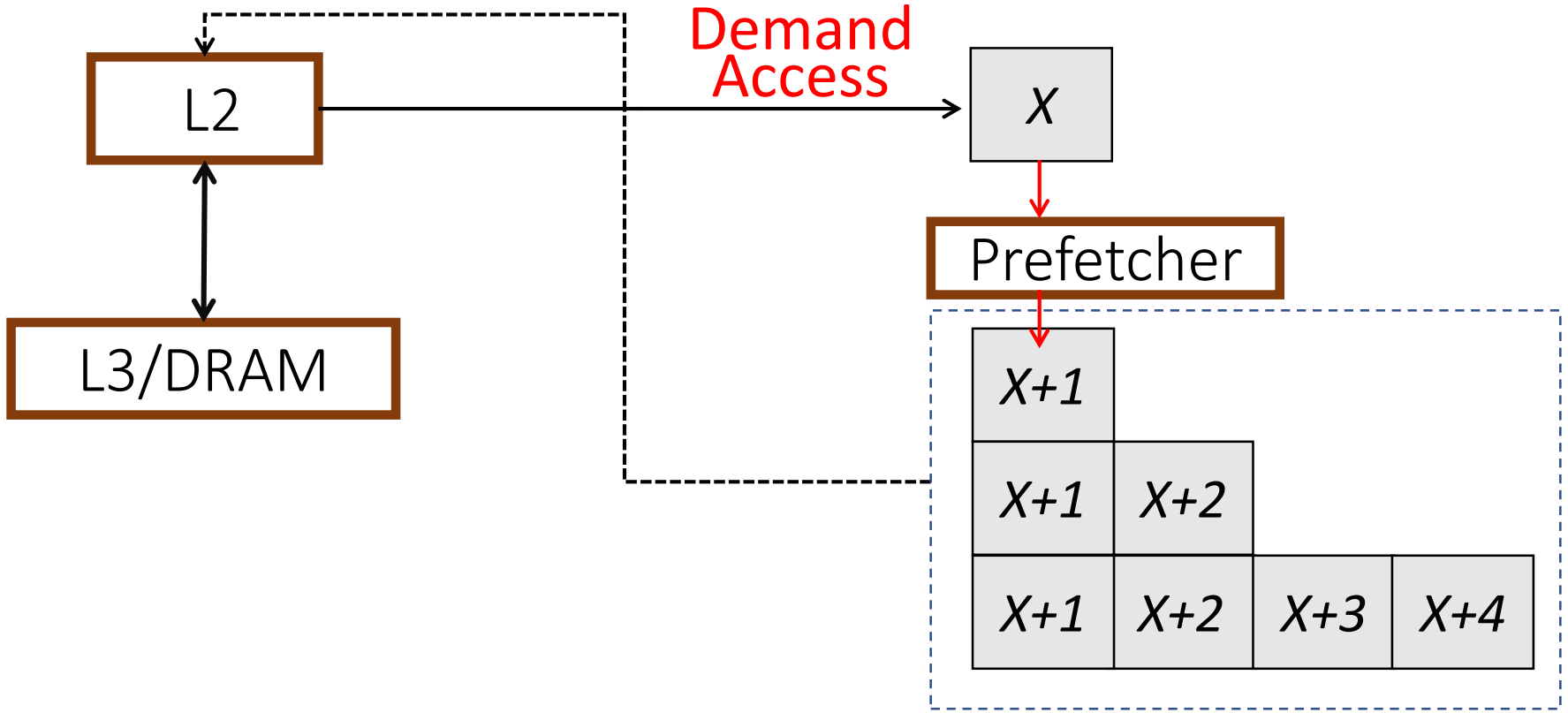
LRU causes thrashing when working set > cache size

Prefetch Engine



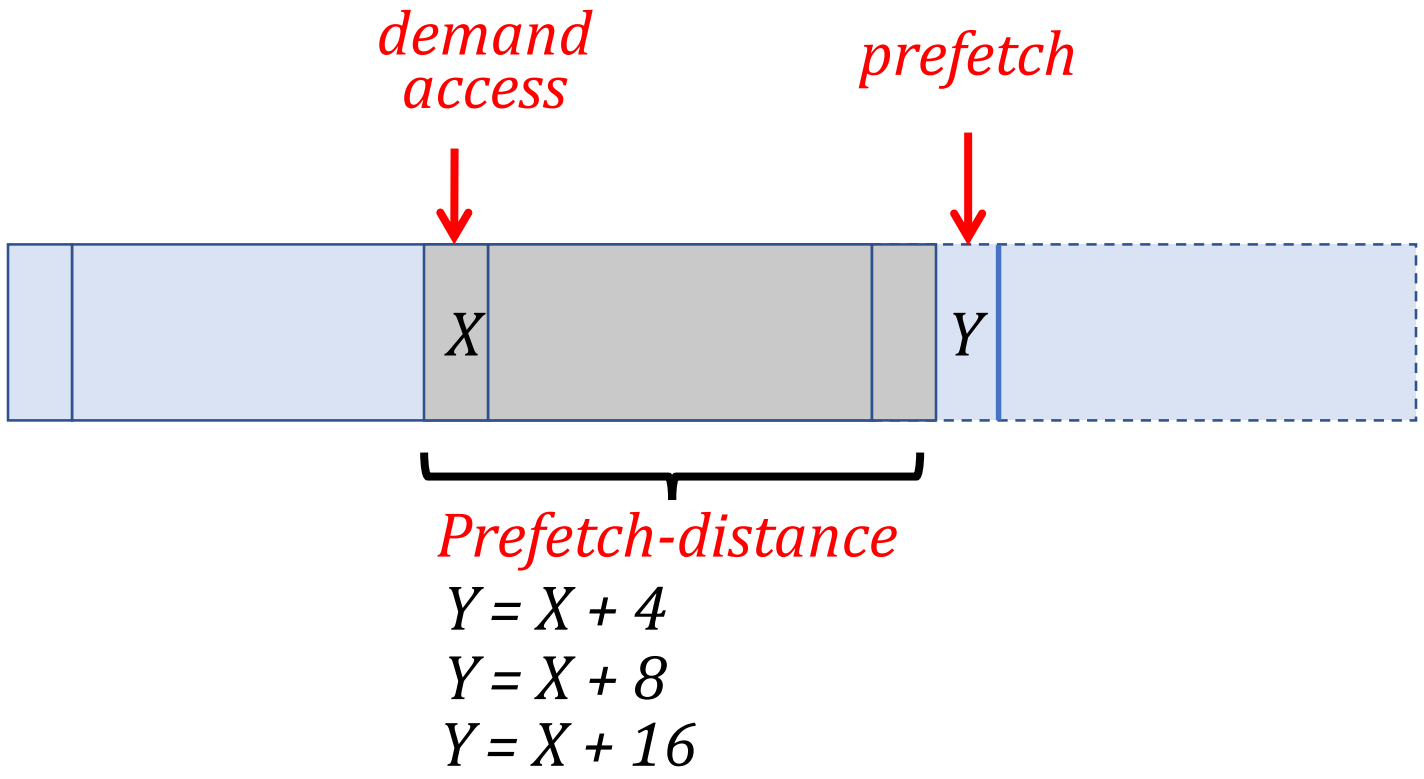
Prefetch Degree

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?

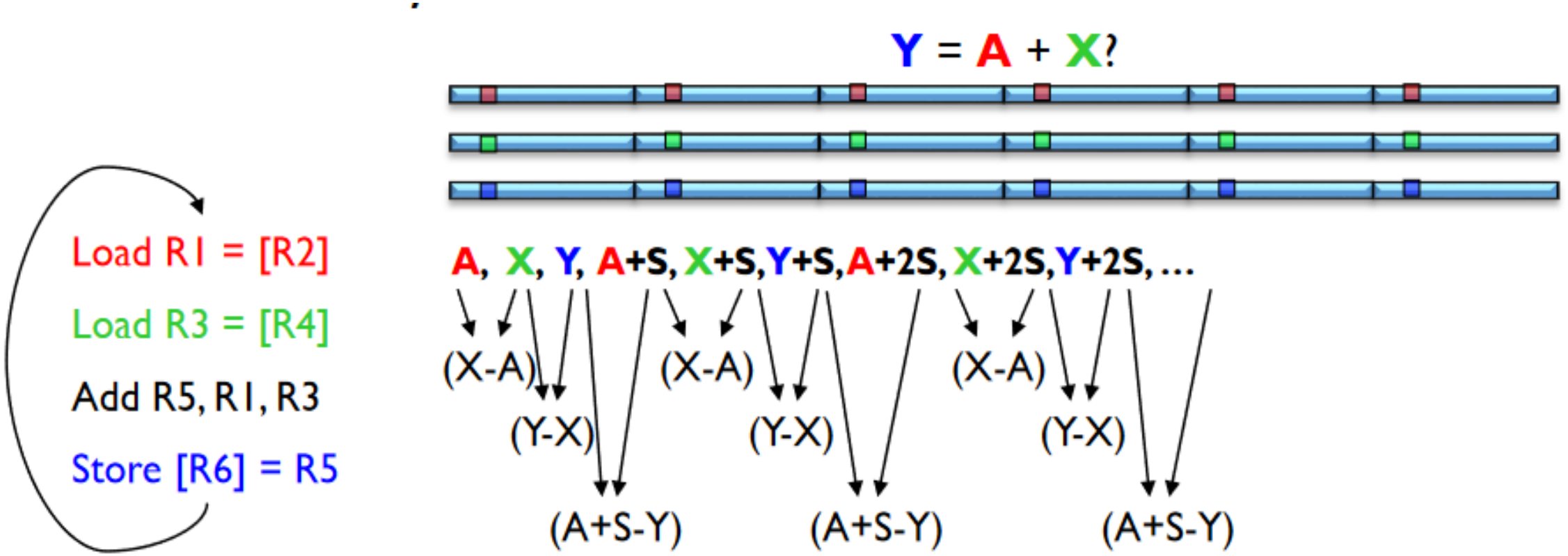


Next-line Prefetcher

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?



Stride Prefetching