# Computer Systems Design

## Lesson 14

## Caching

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

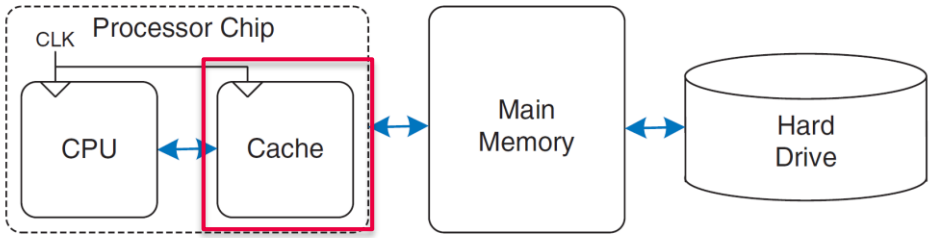
ITMO

# Outline the lesson

- Adaptive memory/data mapping: caches
- Cache associativity
- Cache management protocols
- Virtual/physical indexing/tagging
- Data regeneration vs. storage: trade-off
- Memoization, dynamic instruction reuse

# Adaptive memory/data mapping: caches

Complex processors continuously ***self-adapt data mapping on caches***

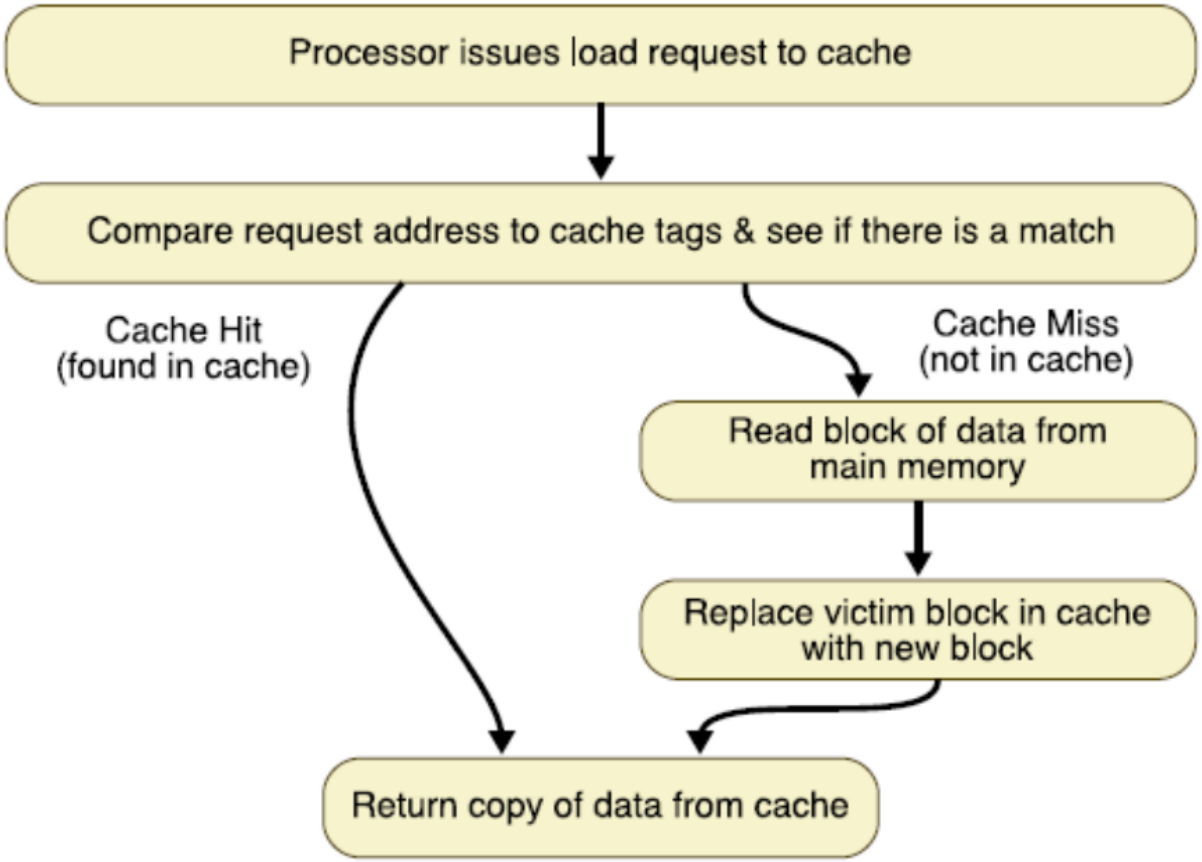



Management issues:

1) *How to **place** data (map on cache banks)?*
2) *How to **update** (write) data?*
3) *How to choose the block for **eviction** (to replace it with new, more useful one)?*

ITMO

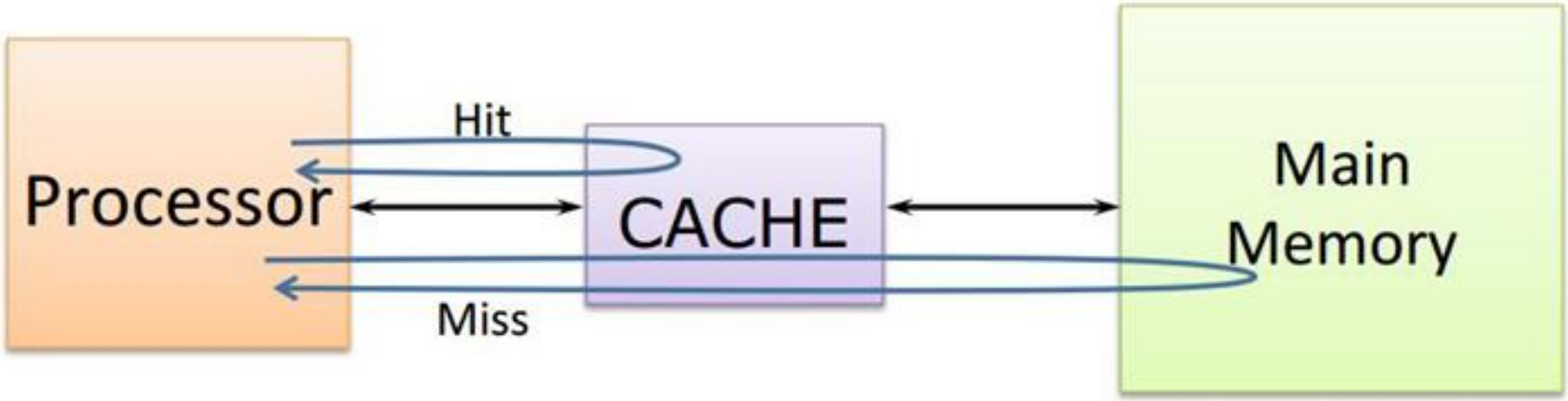
# Cache request algorithm



ITMO

# Cache performance

**Average Memory Access Time (AMAT)** = Hit time + (Miss Rate * Average Miss Penalty)



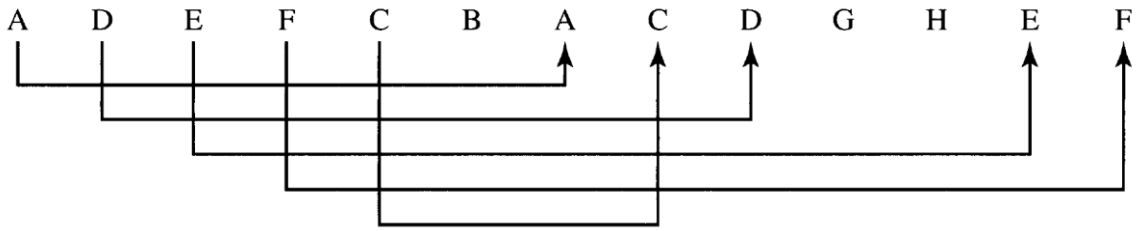
ITMO

# Basic caching heuristics: temporal/spatial locality

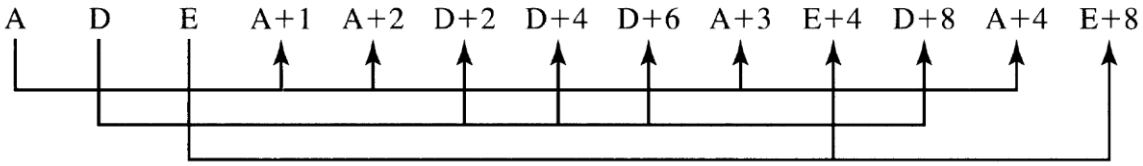
Programmability makes access patterns ***undefined***

… but common (typical) observations can be made:

Temporal locality




Spatial locality




***Strategy: optimize for the most common case***

ITMO

# Cache management protocols

ITMO

# Basics of cache management

Caches operate in blocks/lines (e.g. 64 bytes)

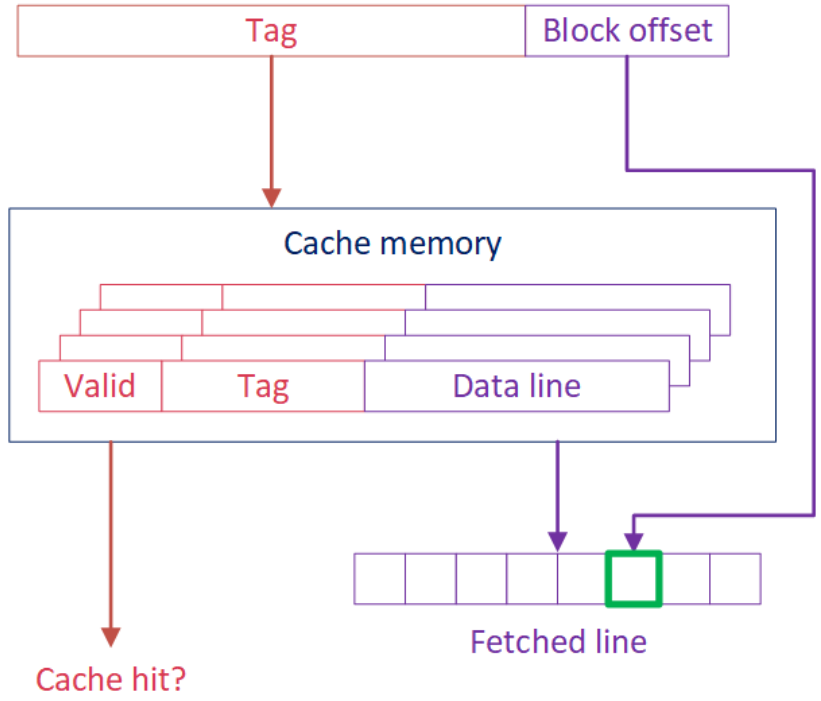
***Only a portion*** of data is placed in cache in each moment of time

Address is split in:

- **Tag**: cache line address

  *used as key to identify line in cache*

- **Cache offset**: offset of data word in a cache line




***Caches can have various internal bank organization with various agility/complexity trade-offs***

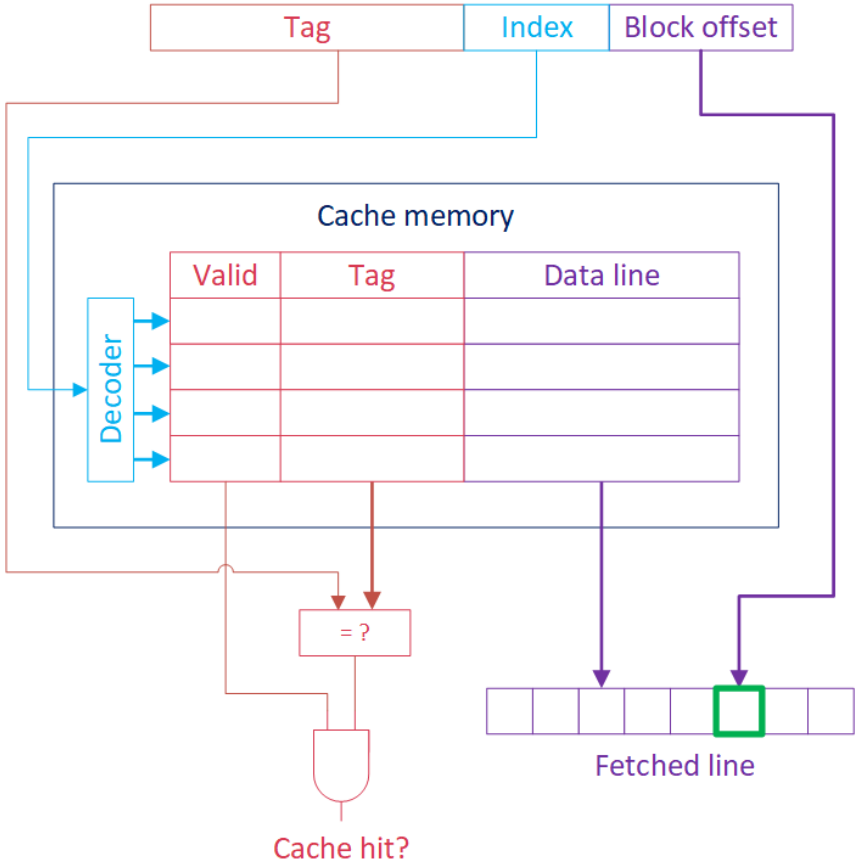
# Cache block placement: direct mapping

*Each block has **single possible placement*** in cache *defined by index*

***Simple fetch**: checking only a single location needed*

***Simple replacement**: one possible eviction candidate*

***Very restrictive**: cache blocks with same Index (but different Tags) **always conflict with each other** (cannot be in cache simultaneously)*



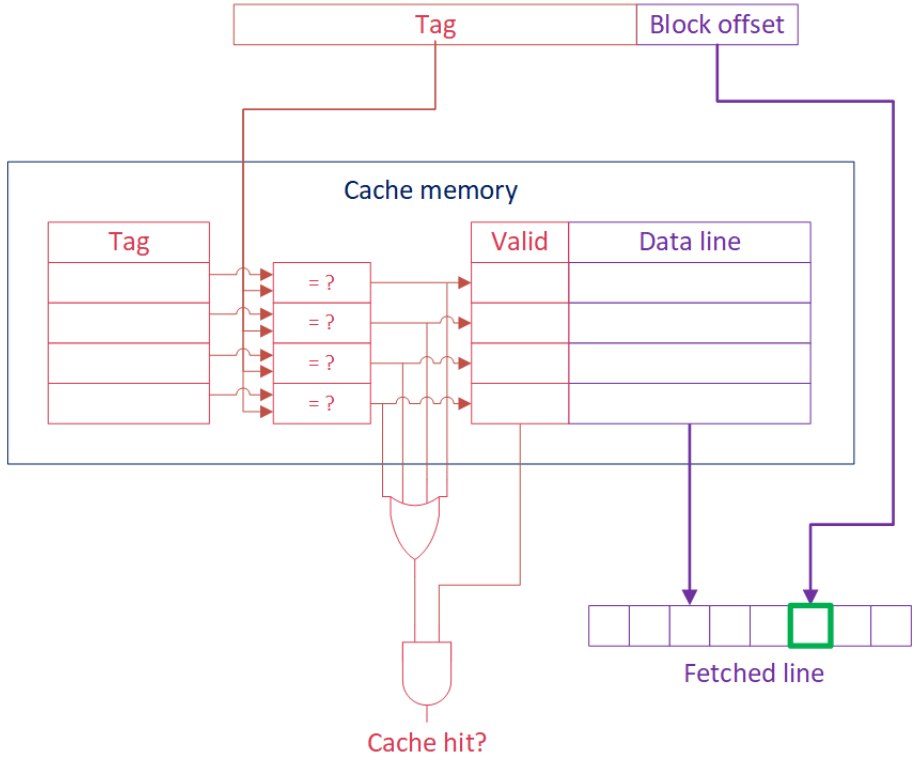
# Cache block placement: fully associative mapping

*Any block can be placed in **any cache location***

***Expensive**: associative memory searching through **all** blocks in cache*

***Eviction:** necessary only when the whole cache is filled, but identification of the least useful block for eviction in the whole cache is needed*

***Very agile**: cache blocks don't conflict until the entire cache memory is filled*



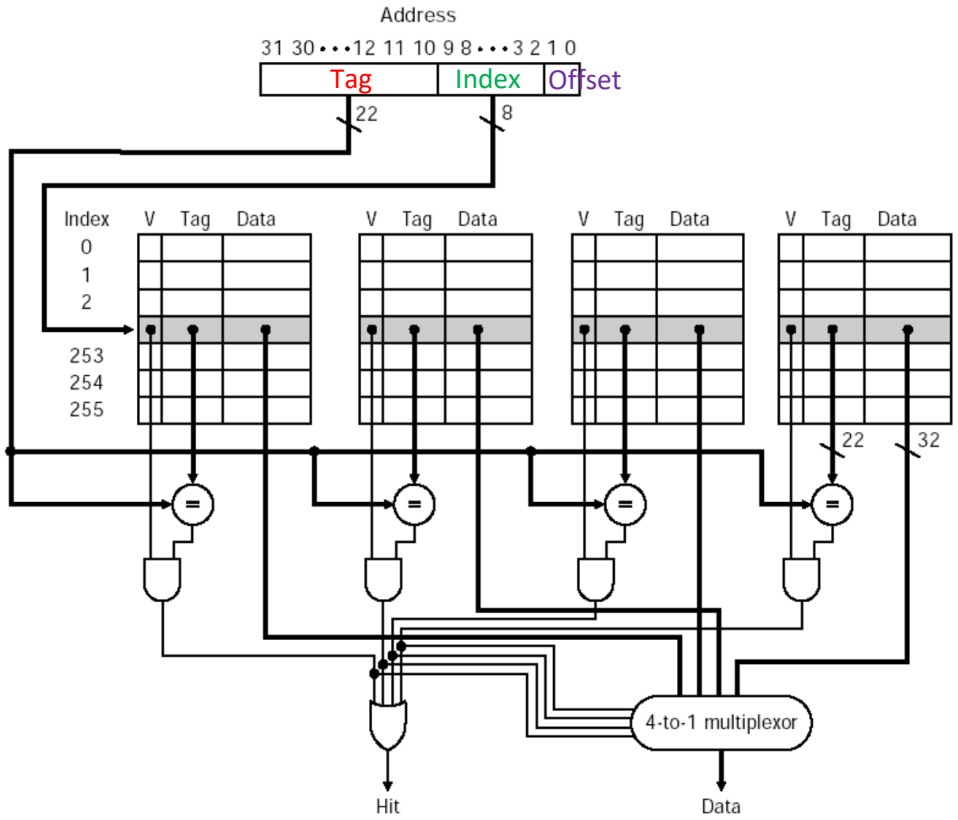
# Cache block placement: set associative mapping

*Each address can be placed **in several locations** in various banks (4-way: 4, 8-way: 8, …)*

*Aliased addresses conflict **only within the set***

***Good complexity/agility trade-off***

***Least useful block for eviction should be identified within the set***




***Commonly used in most cases***

ITMO

# Relations between cache characteristics

| Cache mapping | Address width, bit | Cache block size, bytes | Cache block offset width, bit | Associativity, units | Index width, bit | Tag width, bit | Total blocks in cache, units | Total cache size, bytes |
|---|---|---|---|---|---|---|---|---|
| **Direct** | AW | CBS | CBOW=$\log_2$CBS | 1 | IW | TW=AW-CBOW-IW | CB=2^IW | CS=CB*CBS |
| **Fully associative** | | | | AS | N/A | TW=AW-CBOW | CB=AS | |
| **Set-associative** | | | | AS | IW | TW=AW-CBOW-IW | CB=(2^IW)*AS | |

ITMO

# Cache block write strategies

***Cache hit (data is in the cache):***

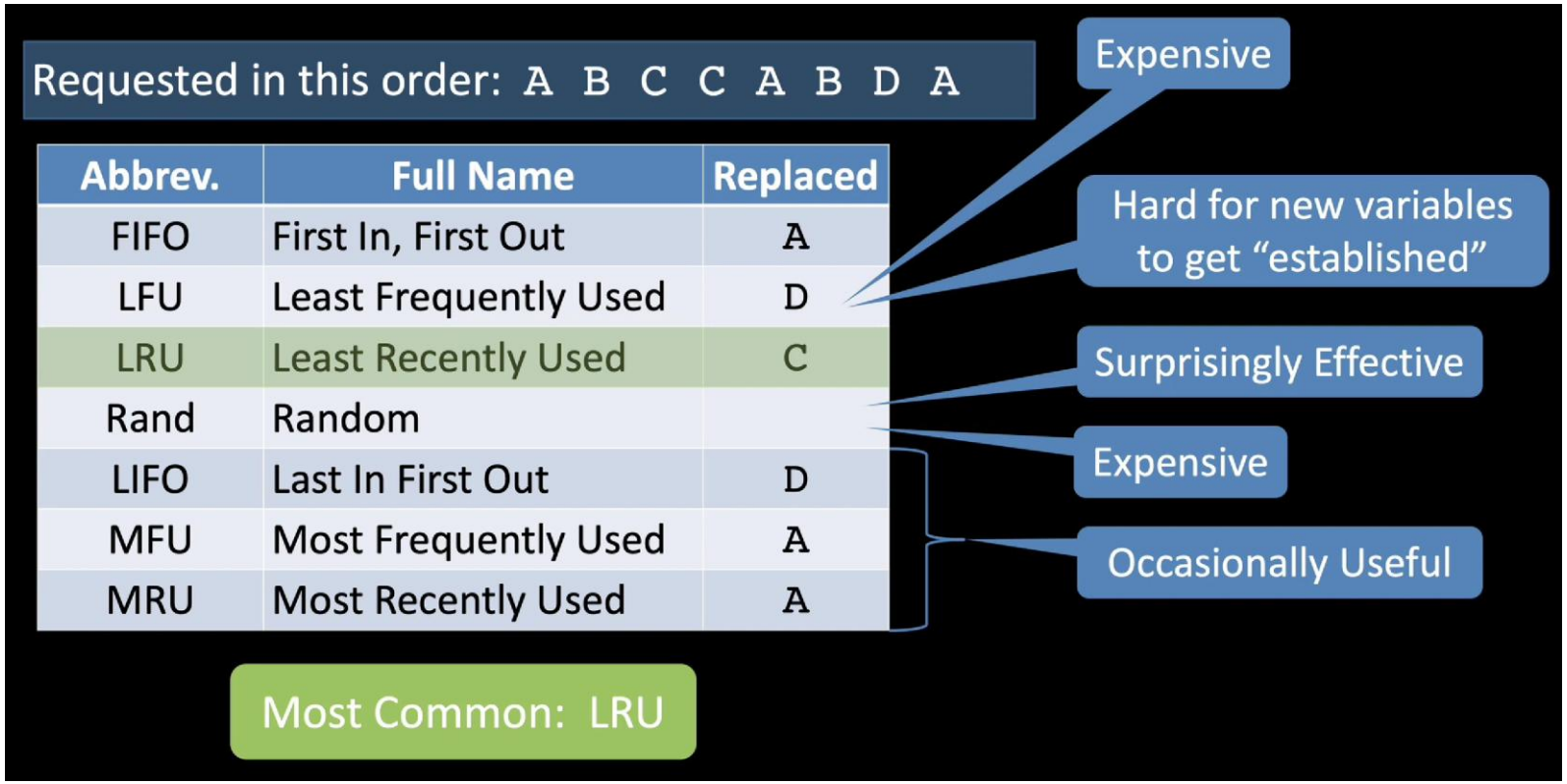
- ***Write-through**: update both cache(s) and main memory*
  *main memory is always in actual state*
  *expensive to write: overwrites cause main memory accesses*
  *cheap to evict: does not need main memory update*
- ***Write-back**: update only cache*
  *main memory holds outdated data*
  *cheap to write: overwrites don't cause main memory accesses*
  *expensive to evict: requires main memory update if written*

***Cache miss (data is not in the cache):***

- ***No write allocate**: update main memory only*
  *cheap to do (less actions)*
  *bad for locality: non-allocated data will likely be needed in the near future*
- ***Write allocate**: allocate written data in cache*
  *expensive to do: more actions, eviction will be likely required*
  *good for locality: allocated data will likely be needed in the near future*

# Cache block eviction strategies

To place a new block in cache, the less useful one should be identified for eviction



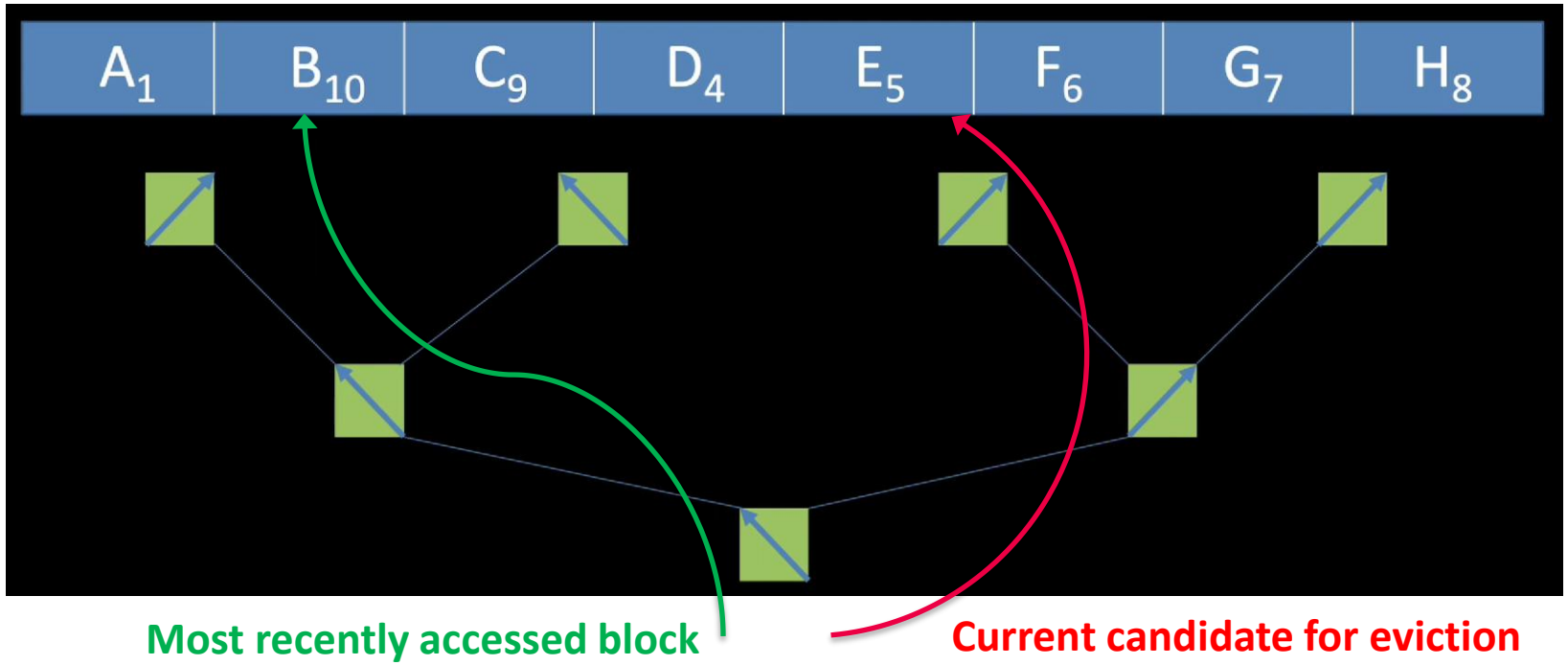

*https://www.youtube.com/watch?v=0qBrbVAJbfc&ab_channel=Prof.Kurmas*

ITMO

# Cache block eviction: Pseudo-LRU algorithm

**Accessing**: continuous construction of ***binary search tree pointing to the last accessed block in the set***

**Eviction**: going through the binary search tree ***in opposite directions***




***Suboptimal but efficient: (n-1) memory, logn computations***

ITMO

# Caching by virtual/physical memory addresses

***Cache search can be parallelized with address translation to reduce latency***

| | Physical indexing | Virtual indexing |
|---|---|---|
| **Physical tagging** | ***PIPT: Physically Indexed, Physically Tagged***<br>*physical address used for both index and tag*<br>*slowest, cache search is done after address translation completes* | ***VIPT: Virtually Indexed, Physically Tagged***<br>*virtual address used for index, physical for tag*<br>*good performance: tag can be fetched in parallel with translation*<br>*aliases are possible, but avoidable*<br>*often used in practice* |
| **Virtual tagging** | ***PIVT: Physically Indexed, Virtually Tagged***<br>*physical address used for index, virtual for tag*<br>*useless: index is needed prior to tag* | ***VIVT: Virtually Indexed, Virtually Tagged***<br>*virtual addresses used for both index and tag*<br>*fastest lookups (no address translation needed for caching)*<br>*bad: aliases (different virtual addresses may refer to the same physical address)* |

ITMO

# Caching computations

# Data regeneration vs. storage trade-off

***Storage requirements can be decreased if data can be recomputed on demand***

***Question: comparative cost of cache lookup and recomputation***

Processing-optimized strategy

- All unique data computed once (in extreme case)
- Larger memory footprint
- Smaller computational workload


***On-demand data regeneration***

***Memoization***

***Dynamic instruction reuse***

Memory-optimized strategy

- Data recomputed on demand
- Smaller memory footprint
- Bigger computational workload

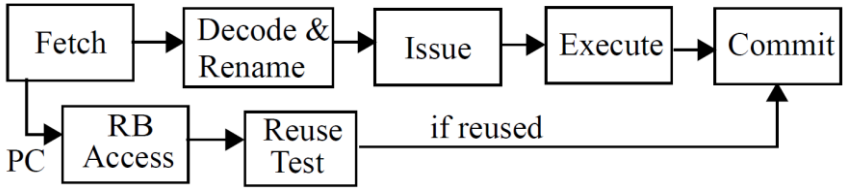
ITMO

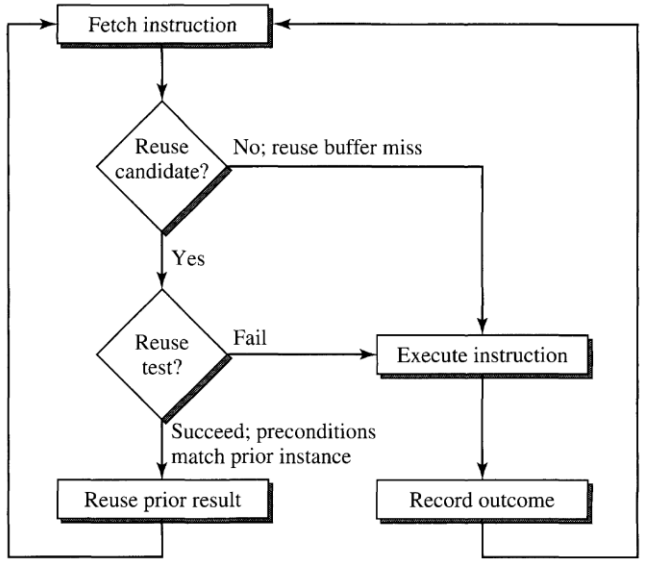
# Caching computations in software: memoization

```
function factorialDynamic() {

    let cache = new Map();

    return function factorial(n) {
        if (cache.has(n)) {
            return cache.get(n);
        } else {
            if (n <= 1) return 1;
            cache.set(n, n * factorial(n-1));
            return cache.get(n);
        }
    }
}
```

# Memoization in CPU microarchitecture: dynamic instruction reuse

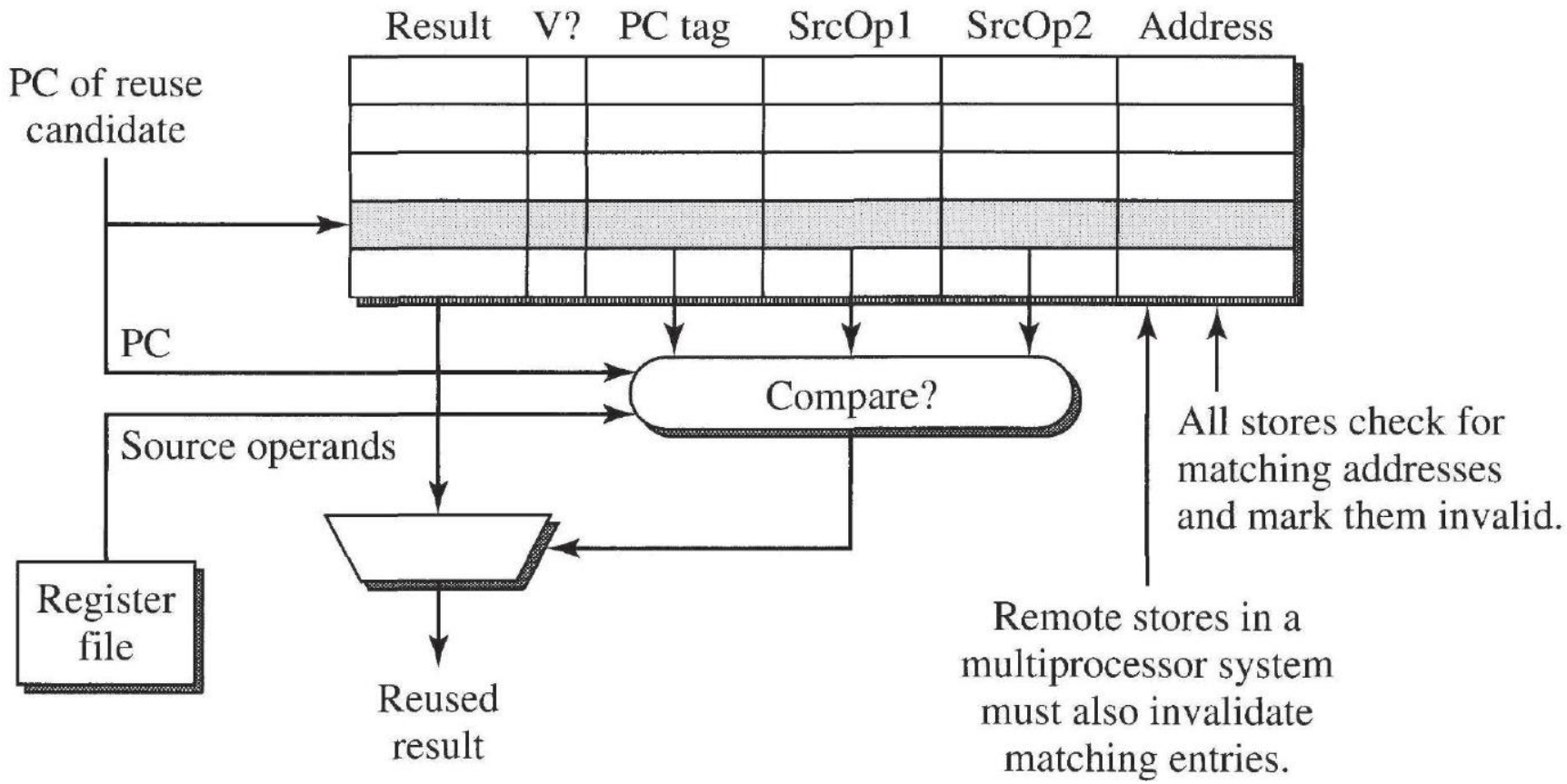
- Research works report ***up to 75%*** of instructions in certain benchmarks provide the same result
- ***Reuse Buffer (RB)*** is allocated for instruction and the result it produced
- Reuse test checks ***identity of operands***. If success – instruction goes straightly to commit
- ***RAW dependency effect reduced***



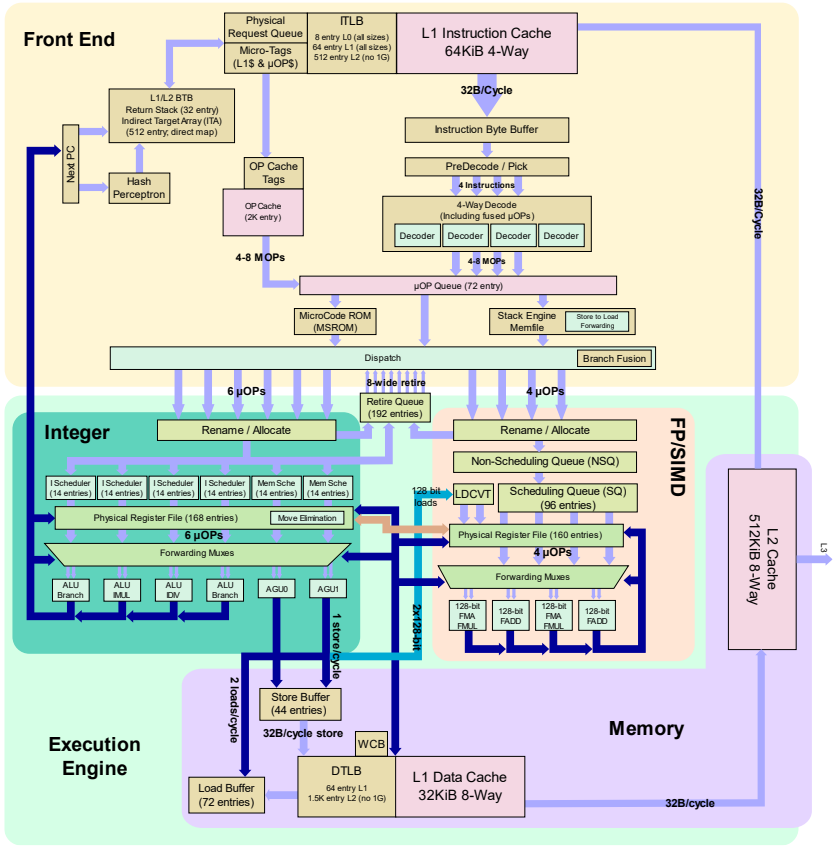




*Avinash Sodani, Gurindar S. Sohi. Understanding the differences between value prediction and instruction reuse. MICRO 31, 1998*

# Dynamic instruction reuse: RB structure



# Caches in modern CPU cores?




***AMD Zen***

*https://en.wikichip.org/wiki/amd/microarchitectures/zen*

# Caches in modern CPU cores

*ITLB: instructions page address translations*

*BTBs: branch targets*

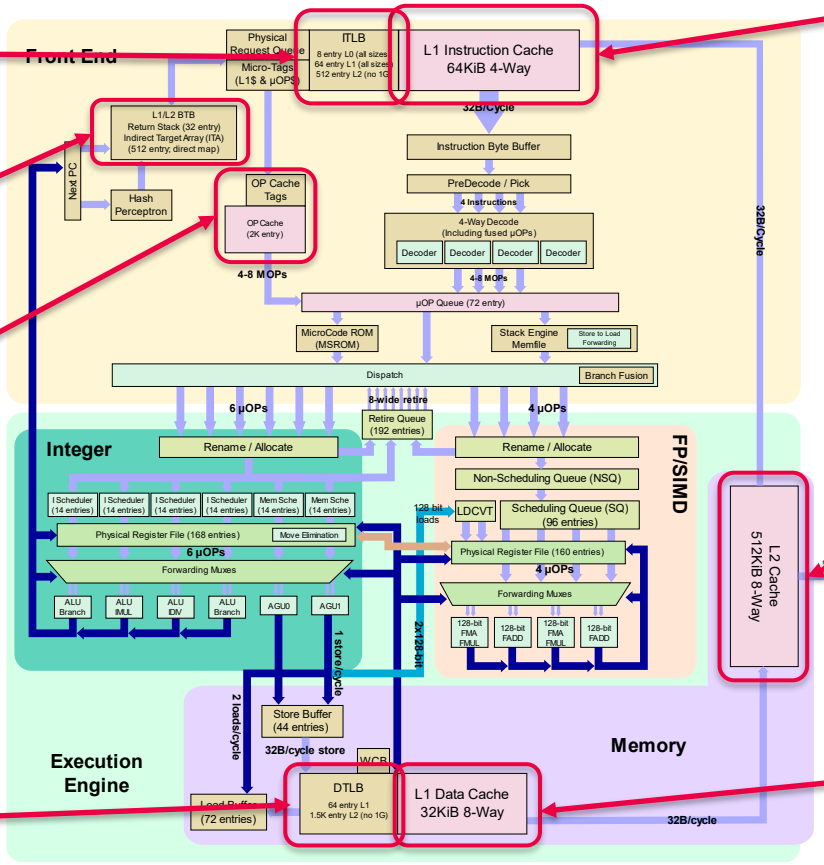
*OP cache: decoded instructions*

*L1 cache: instructions*

*L2 cache: instructions and data*

*L1 cache: data*

*DTLB: data page address translations*




***AMD Zen***

*https://en.wikichip.org/wiki/amd/microarchitectures/zen*

# Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025