18. Cache organization: Performance & Tradeoffs

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

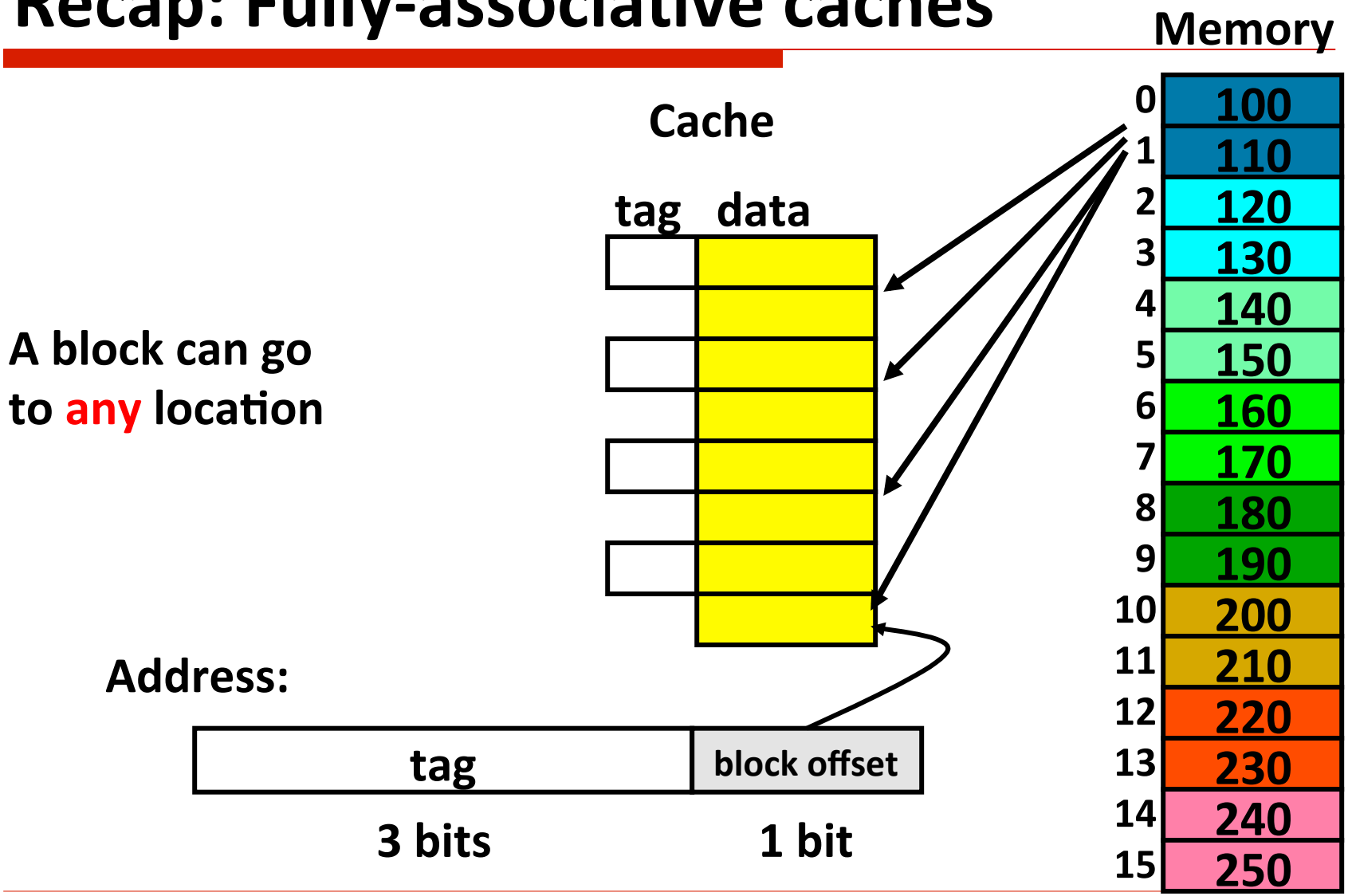
EECS Department
University of Michigan in Ann Arbor, USA

Recap: Fully-associative caches

- We designed a fully associative cache.
  - Any memory location can be copied to any cache line.
  - We check every cache tag to determine whether the data is in the cache.

- This approach can be too slow sometimes.
  - Parallel tag searches are expensive and can be slow. Why?

Recap: Fully-associative caches

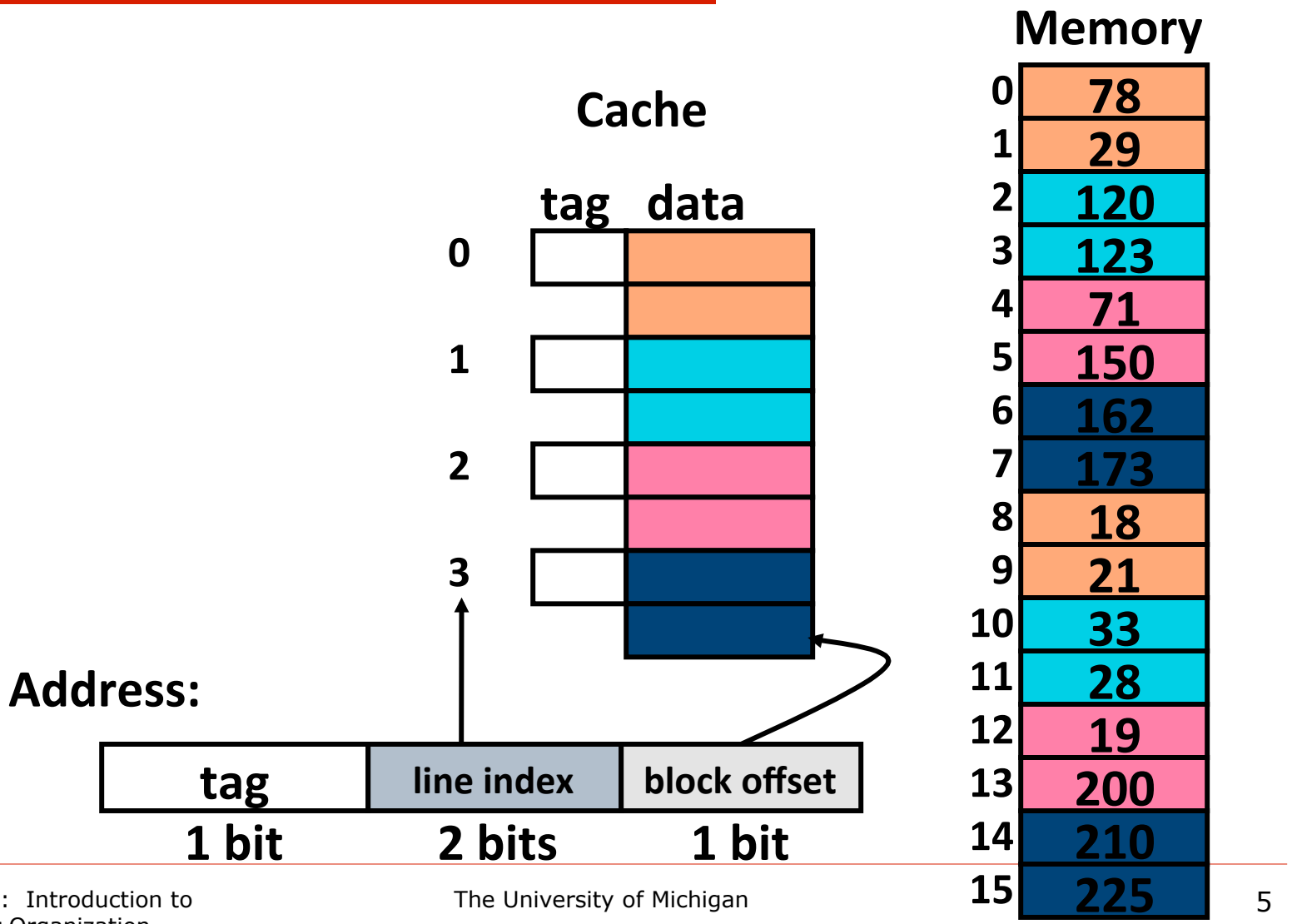


Recap: Direct-mapped caches

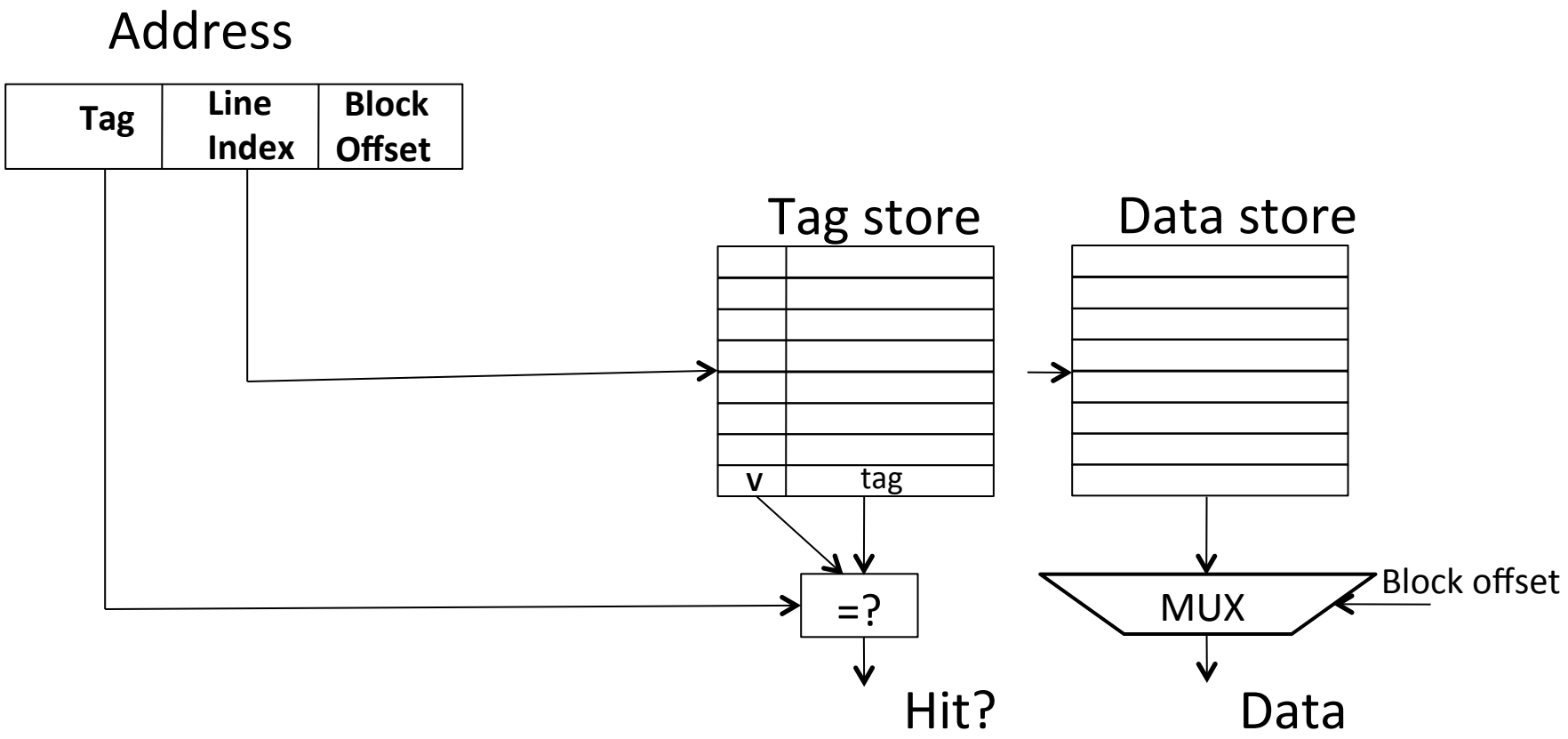
- We can redesign the cache to eliminate the requirement for parallel tag lookups.
  - Direct mapped caches partition memory into as many regions as there are cache lines.
  - Each memory region maps to a single cache line in which data can be placed.
  - You then only need to check a single tag – the one associated with the region the reference is located in.
- Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time

  One index → one entry

Recap: Direct-mapped caches



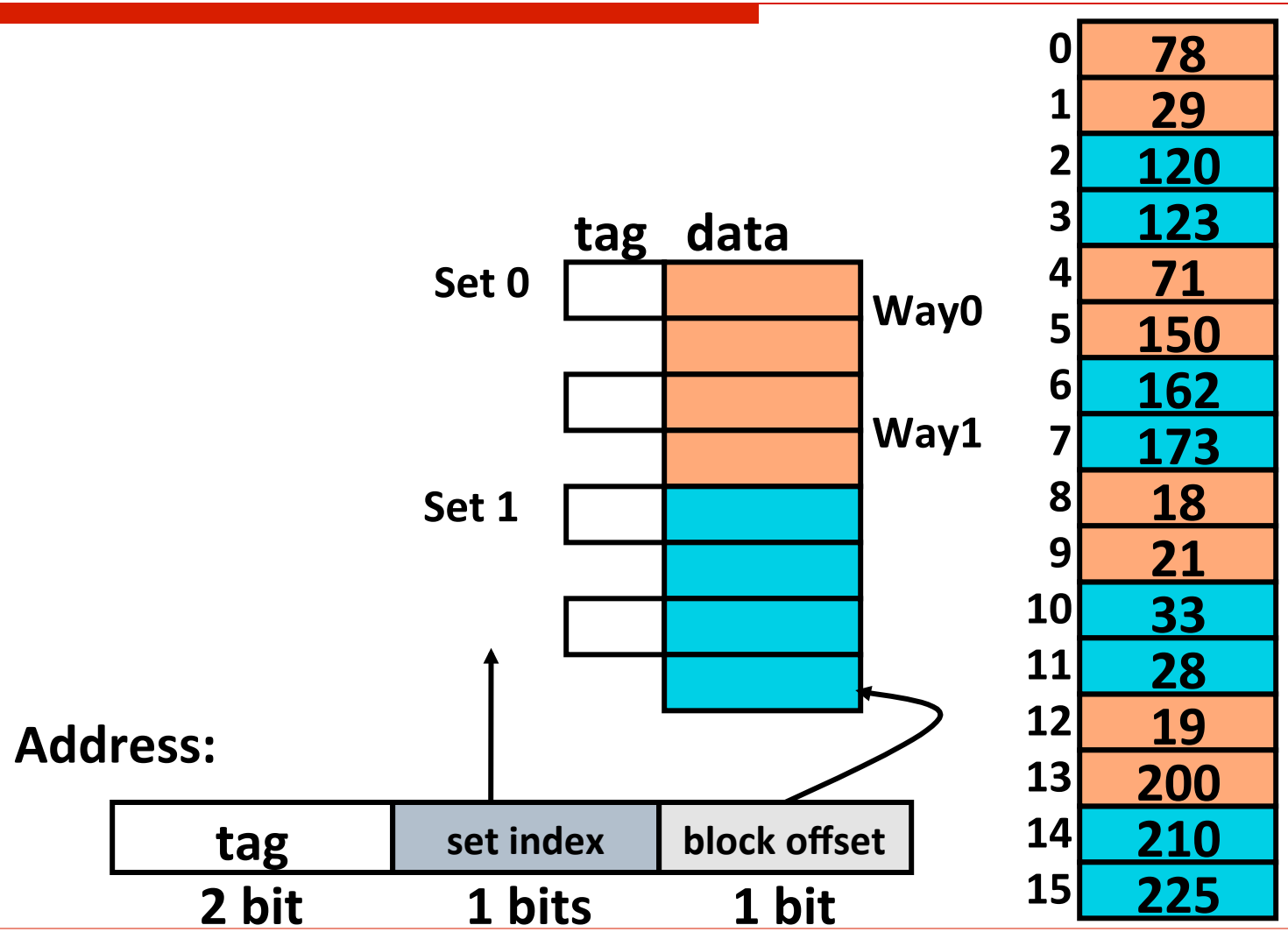
Direct-Mapped Cache: Placement and Access



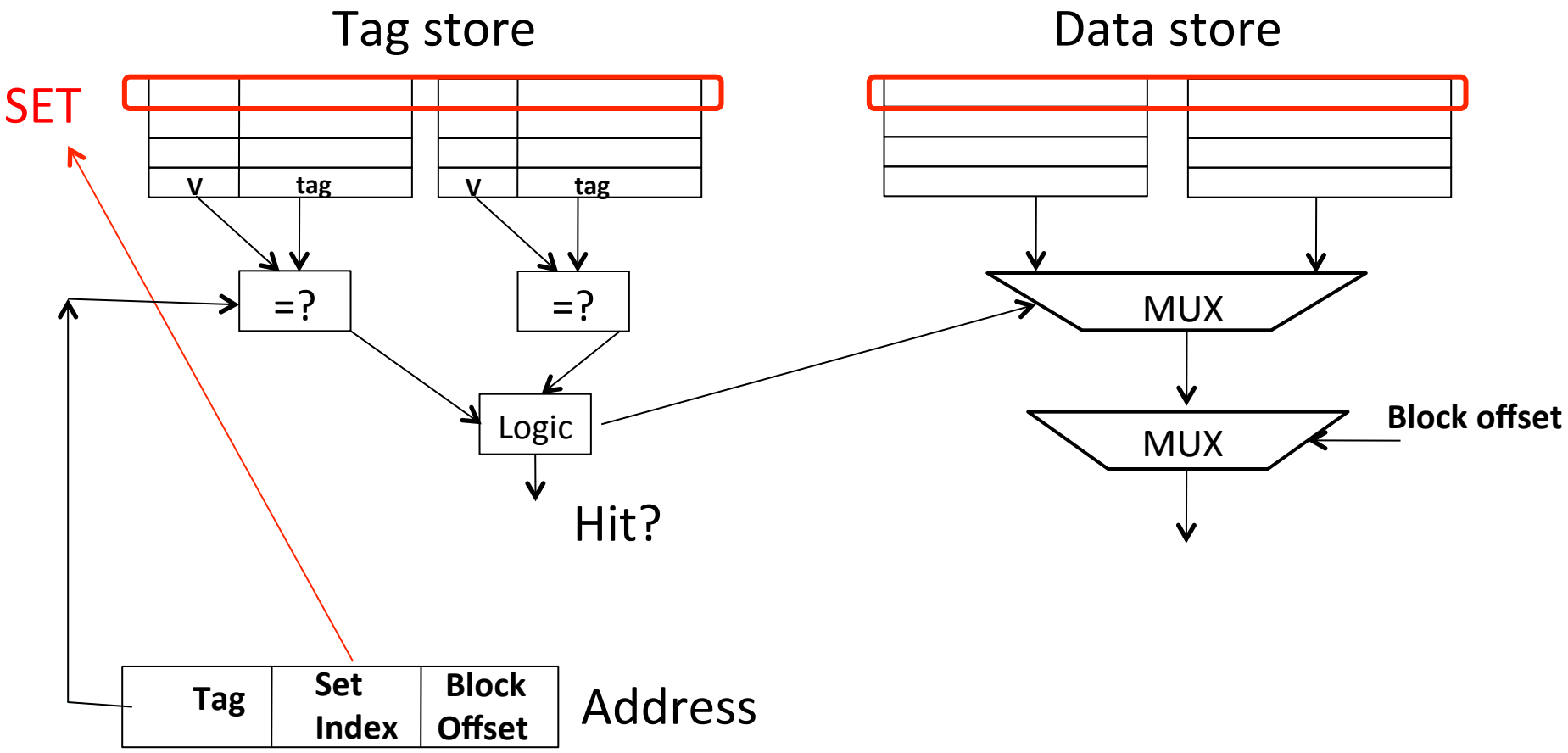
Recap: Set associative caches

- Set associative caches:
  - Partition memory into regions
    - like direct mapped but fewer partitions
  - Associate a region to a set of cache lines
    - Check tags for all lines in a set to determine a HIT
- Treat each line in a set like a small fully associative cache.
  - LRU (or LRU-like) policy generally used.

Recap: Set-associative cache



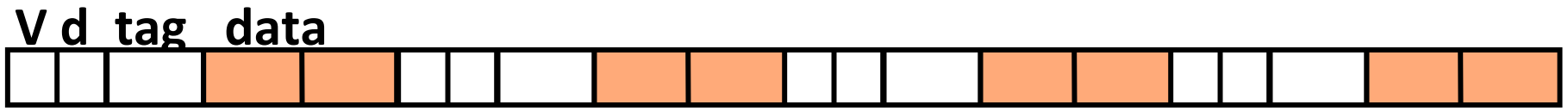
Set-Associative Cache: Placement and Access



Cache Organization Comparison

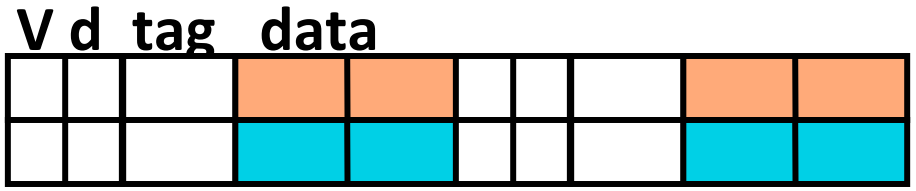
Block size = 2 bytes, total cache size = 8 bytes for all caches

1. Fully associative (4-way associative)



2. Direct mapped

3. 2-way associative



Reasons for cache misses: a.k.a. *the 3 C's of cache misses*

- First reference to an address
  - **Compulsory** miss
    - Also sometimes called a **"cold start"** miss
    - First reference to any block will always miss
- Cache is too small to hold all the data
  - **Capacity** miss
    - Would have had a hit with a large enough cache
- Replaced it from a busy set
  - **Conflict** miss
    - Would have had a hit with a fully associative cache

Classifying cache misses

- Can you classify all cache misses?
  - Compulsory miss?
  - Capacity miss?
  - Conflict miss?
- Yes! (with a cache simulator)
  - Simulate with a cache of unlimited size (cache size = memory size) - Any misses must be compulsory misses
  - Simulate again with a fully associative cache of the intended size - Any new misses must be capacity misses
  - Simulate a third time, with the actual intended cache - Any new misses must be conflict misses

Fixing cache misses

- **Capacity** misses
  - Would have had a hit with a large enough cache
  - Reduce by building a bigger cache
- **Conflict** misses
  - Would have had a hit with a fully associative cache
  - Cache does not have enough associativity
  - Reduce by increasing associativity
- **Compulsory** misses
  - First reference to a address
  - No way to completely avoid these
  - Reduce by increasing block size
  - This reduces the total number of blocks

Class Problem 2

Given the following cache configuration:

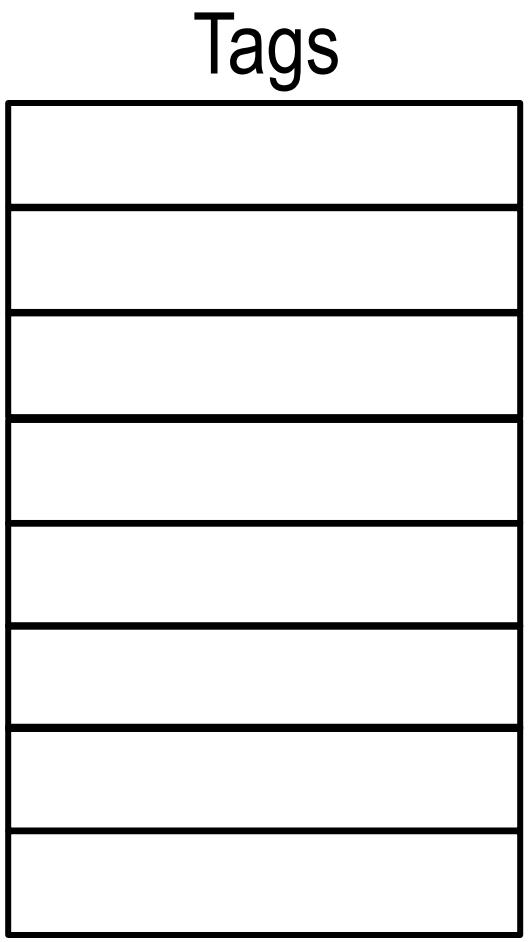
Cache size = 16 bytes
Block size = 2 bytes
Direct mapped

Write allocate
LRU replacement policy
Address size = 16 bits,
byte addressable memory

For the following sequence of addresses, label each reference as a hit (H) or miss (M). For the misses, indicate the type (conflict, capacity, compulsory).

4 7 21 22 23 20 5 7 36 6 4

Class Problem 2



4
7
21
22
23
20
5
7
36
6
4

Class Problem 2

4	7	21	22	23	20	5	7	36	6	4

First accesses

4 7 21 22 36

1. Compulsory – 4 7 21 22 36 ➜ 5 misses

Fully Associative

M M M M H H H H M H H ➜ 5 misses

2. Capacity – Fully Associative (5) – Compulsory (5) ➜ 0 misses

Direct mapped

M M M M H H M M M H M ➜ 8 misses

3. Conflict – Direct Mapped (8) – Fully Associative (5)

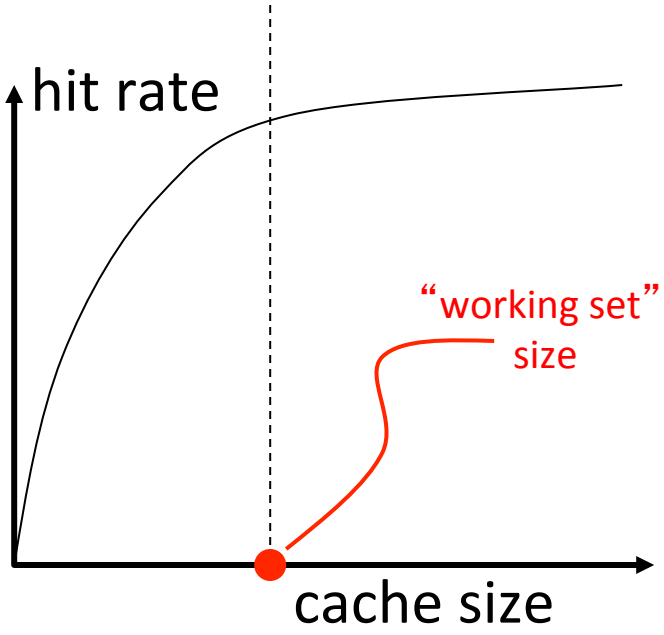
➜ 8 – 5 ➜ 3 misses

Cache Parameters vs. Miss Rate

- Cache size
- Block size
- Associativity
- Replacement policy

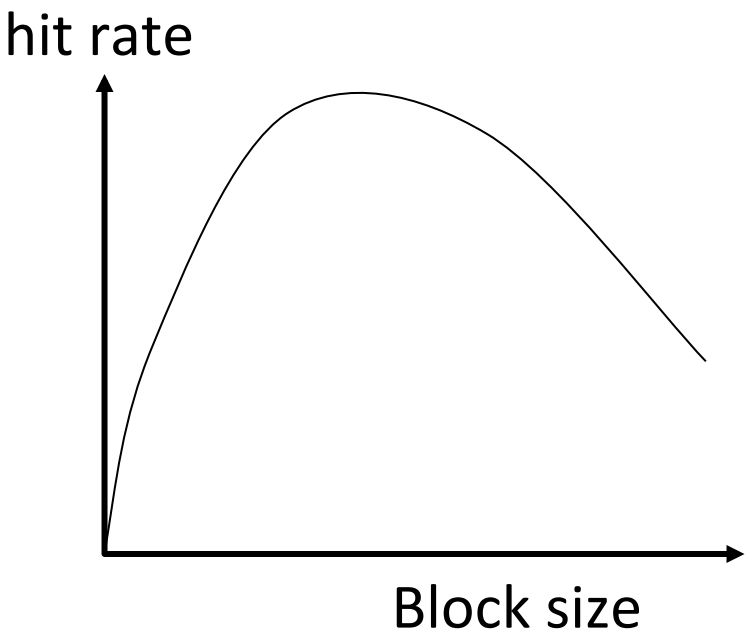
Cache Size

- Cache size in the 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 executing application references
  - **Within a time interval**



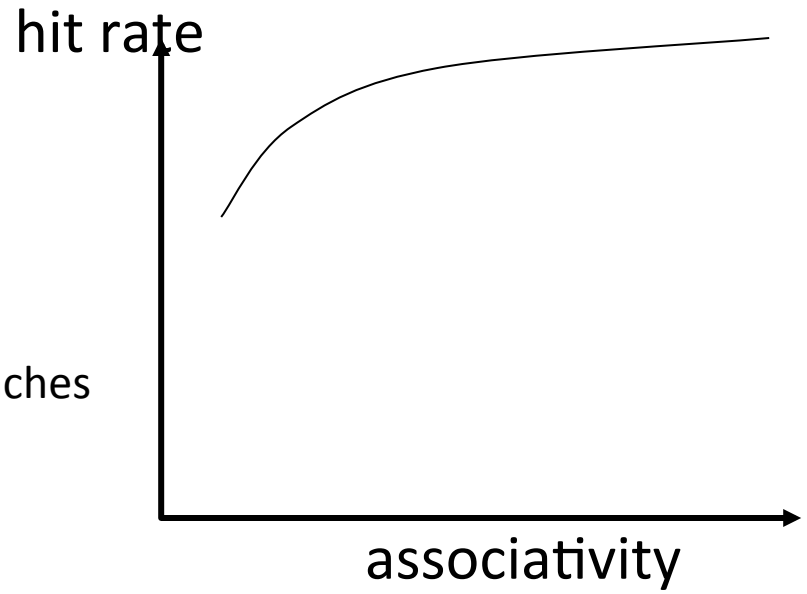
Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each with V bit)
      - Can improve "write" performance
- 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



Associativity

- How many blocks can map to the same index (or set)?
- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns
- Smaller associativity
  - lower cost
  - faster hit time
    - Especially important for L1 caches
- Power of 2 associativity?



Class Problem 3

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 - Miss

0x30f - Miss

0x510 - Miss

0x31f - Hit

0x72d - Miss

0x72f - Hit

0x320 - Miss

0x520 - Miss

0x720 - Miss

Block size: ?

Associativity: ?

Number of sets: ?

Class Problem 3 Answer

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss

0x30f – Miss

0x510 – Miss

0x31f – Hit

0x72d – Miss

0x72f – Hit

0x320 – Miss

0x520 – Miss

0x720 - Miss

<u>Determine block size</u>

First hit must be brought in by another miss

Take closest address: 0x310, so know block size must be at least 16 bytes so 0x31f brought in when 0x310 miss occurs

Now, is the block size larger? Know that 0x30f was a miss, thus 0x310 and 0x30f not in the same block. Thus, block size must be <= 16 bytes

Thus Block Size = 16 bytes

Class Problem 3 Answer (2)

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss 

0x30f – Miss

0x510 – Miss 

0x31f – Hit 

0x72d – Miss

0x72f – Hit 

0x320 – Miss 

0x520 – Miss 

0x720 - Miss 

Determine associativity

Assume direct mapped: 3-bit tag, 5-bit index, 4-bit offset.
If direct mapped 0x310 and 0x510 would both map to index 9,
Thus 0x31f could not be a hit. So, not direct mapped.

Assume 2-way associative: 4-bit tag, 4-bit index, 4-bit offset
This fixes the green accesses, and allows 0x31f to be a hit.

What about > 2-way associative?
Now we also know that 0x720 is a miss even though 3 accesses earlier 0x72f was a hit, and thus it is in the cache. The intervening 2 accesses must kick it out, 0x320 and 0x520. Both go to set 2. If the associativity was > 2, then 0x720 would be a hit. So, must conclude that cache is 2-way associative.

Lastly, number of sets = 512 / (2 * 16) = 16

Recap: Cache for Instructions?

- Instructions should be cached as well.
- We have two choices:

  1. Treat instruction fetches as normal data and allocate cache lines when fetched.
  2. Create a second cache (called the instruction cache or ICache) which caches instructions only.

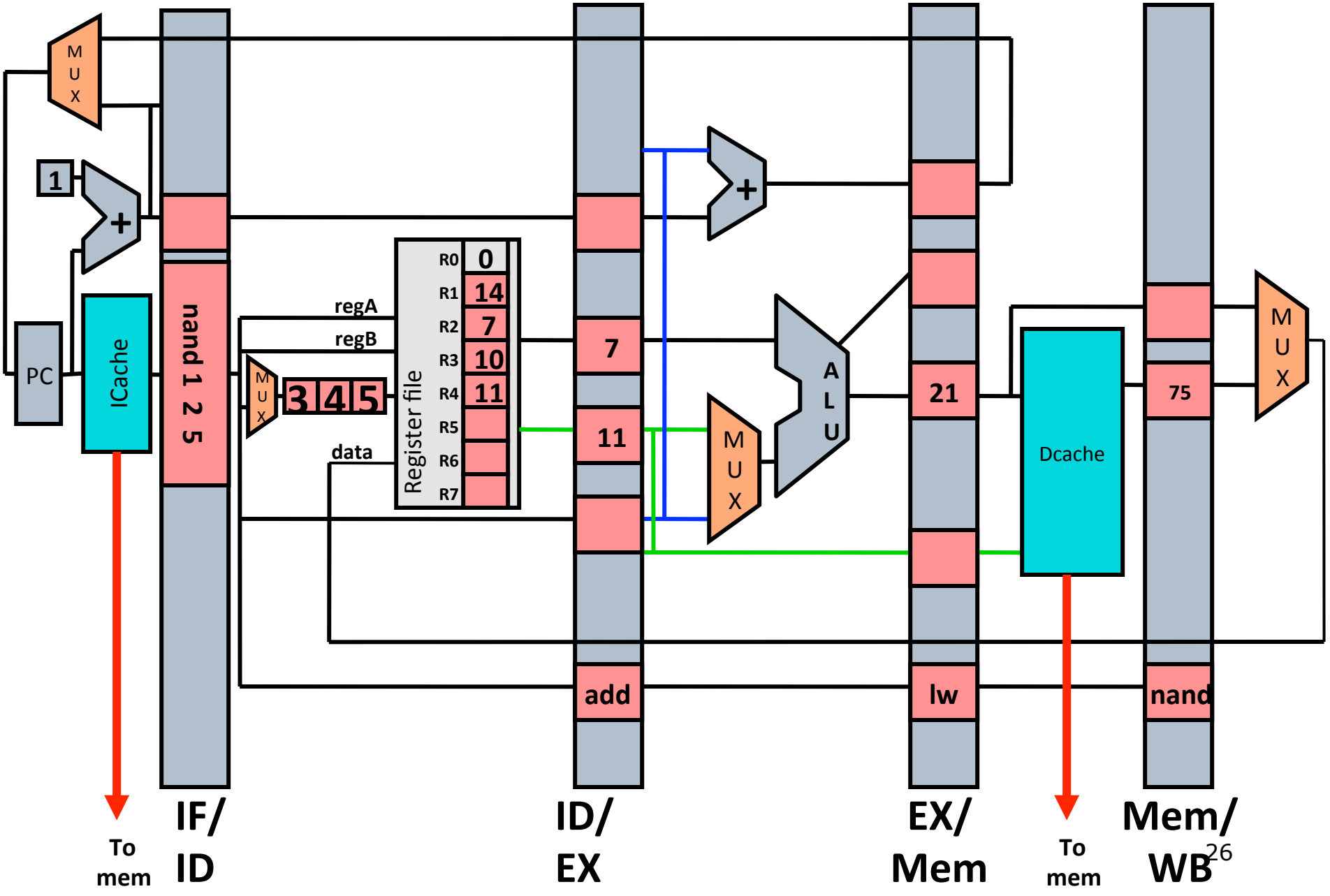
     How do you know which cache to use?

     What are advantages of a separate ICache?

Integrating Caches into a Pipeline

- How are caches integrated into a pipelined implementation?
  - Replace instruction memory with Icache
  - Replace data memory with Dcache
- Issues
  - Memory accesses now have variable latency
  - Both caches may miss at the same time

LC2K Pipeline with Caches



Putting It Together & More

- What's inside modern processors?
  - Deep memory hierarchy
  - Processors which can hide memory latency
  - More and more parallelism!
  - Lets take a peak

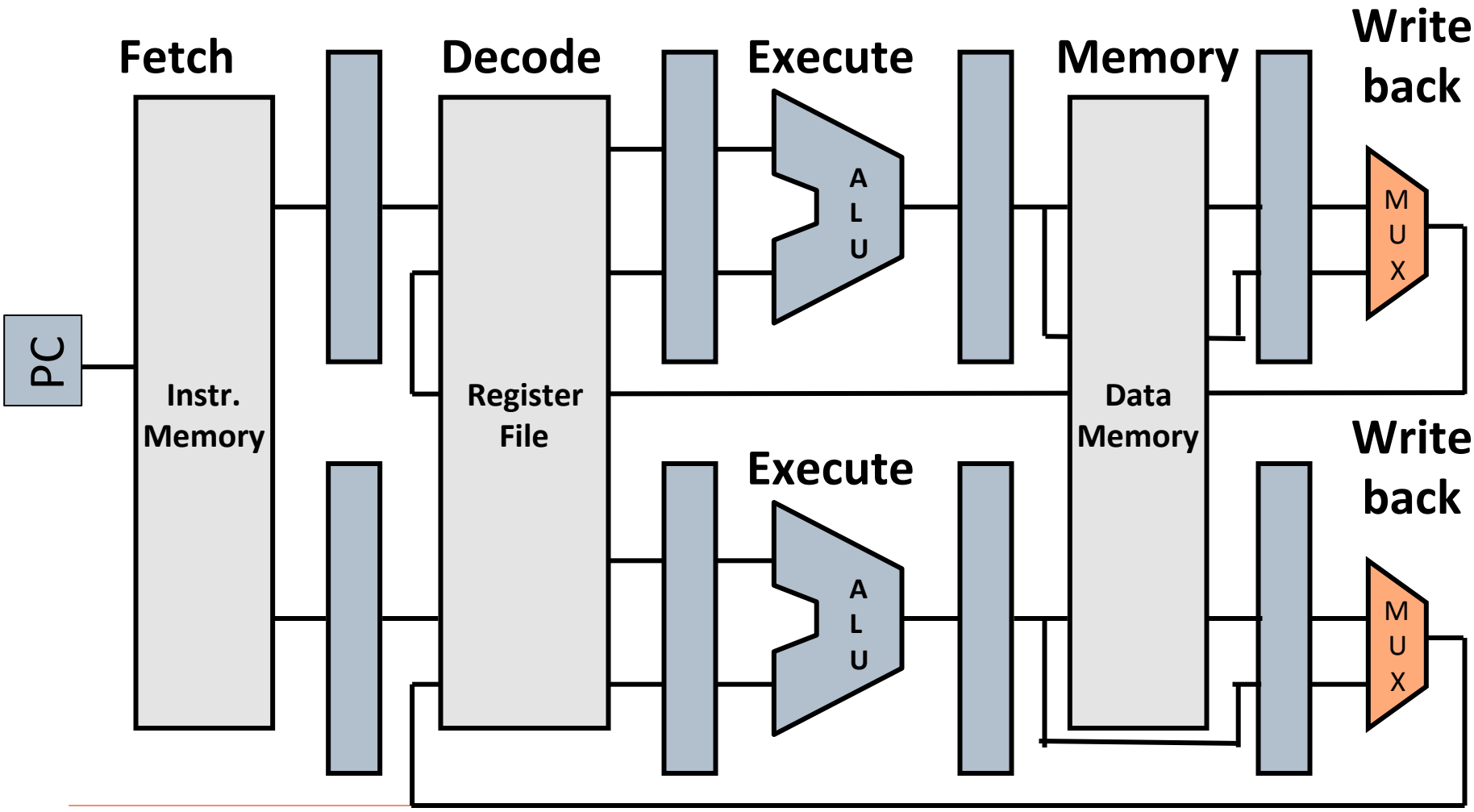
Building with Pipelines

- CPI for pipelining:
  - 1 (ideal case - no stalls)
  - > 1 (reality, depends on program)
- What if we want to improve performance more?
  - Want CPI as low as possible – lower than 1
- Use Parallelism
  - Instruction Level Parallelism (ILP) – Within task
  - Thread Level Parallelism (TLP) – Having many tasks
  - Data Level Parallelism (DLP) – Many tasks with same instructions

Creating more pipelines

- Instruction Level Parallelism – Superscalar Pipeline
  - Have two or more pipelines in same processor
  - pipelines need to work in tandem to improve single program performance
- Thread Level Parallelism – Multi-core
  - Have two or more processors (Independent Pipelines)
  - Need more programs or a parallel program
  - does not improve single program performance
- Data Level Parallelism – Single Inst. Multiple Data (SIMD)
  - Have two or more execution pipelines (ID->WB)
  - Share the same fetch and control pipeline to save power (IF+cont.)
  - Similar to GPU's

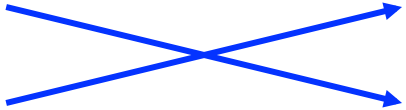
ILP Techniques: Superscalar



Other Techniques for ILP: Out of Order Execution

- Eliminating stall conditions decreases CPI
- Reorder instructions to avoid stalls
- Example (5-stage LC2K pipeline):

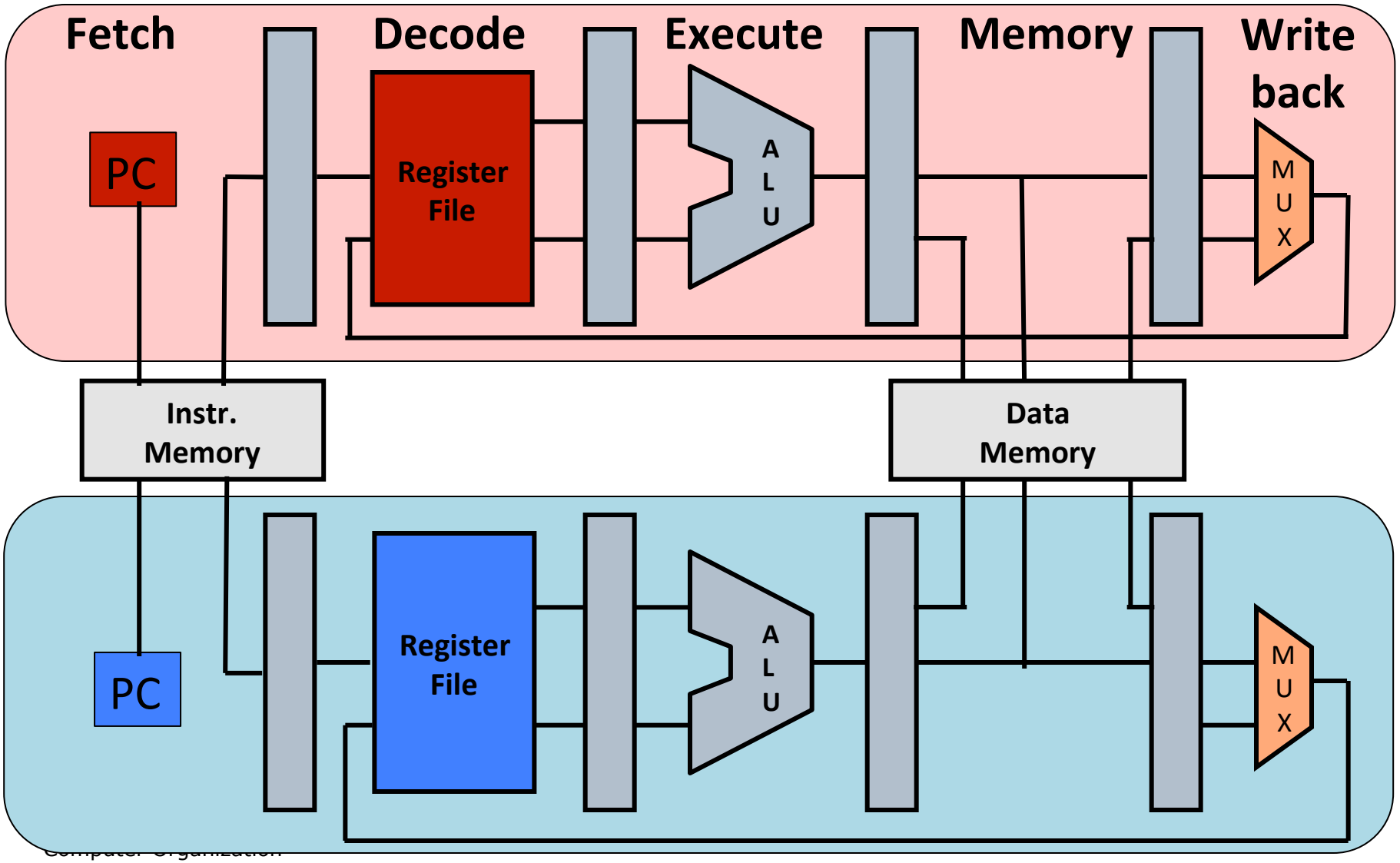
```
add   1 2 3          add   1 2 3
lw    3 2 16         lw    3 2 16
nor   2 6 7          add   4 5 1
add   4 5 1          nor   2 6 7
```



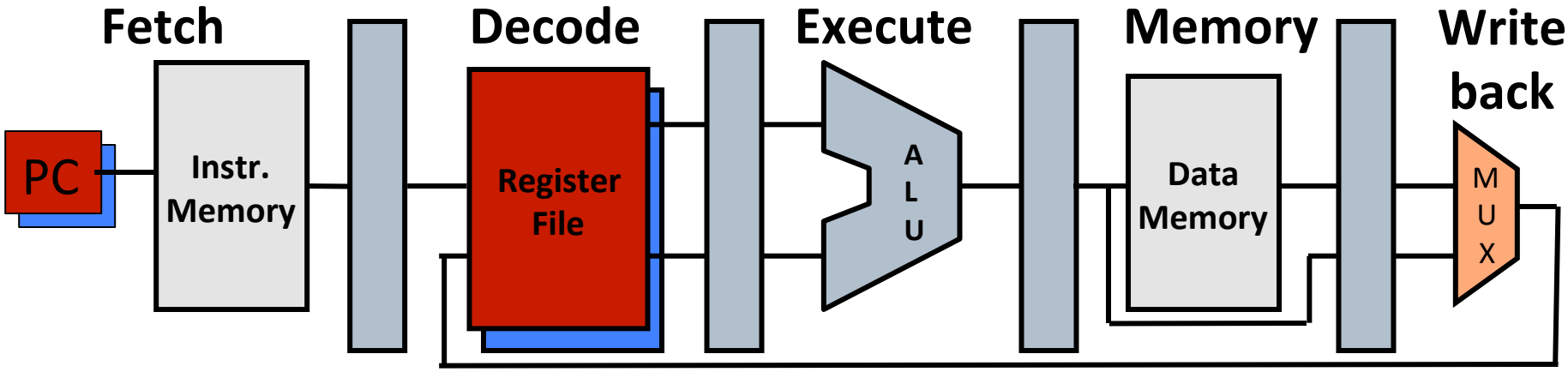
Why Use Out of Order Execution?

- Some instructions take a long time to execute
  - Floating point operations
  - Some loads and stores (more when we talk about memory hierarchy)
- Options:
  - Increase cycle time
  - Increase number of pipeline stages
  - Execute other instructions while you wait

TLP Techniques: Multiprocessors

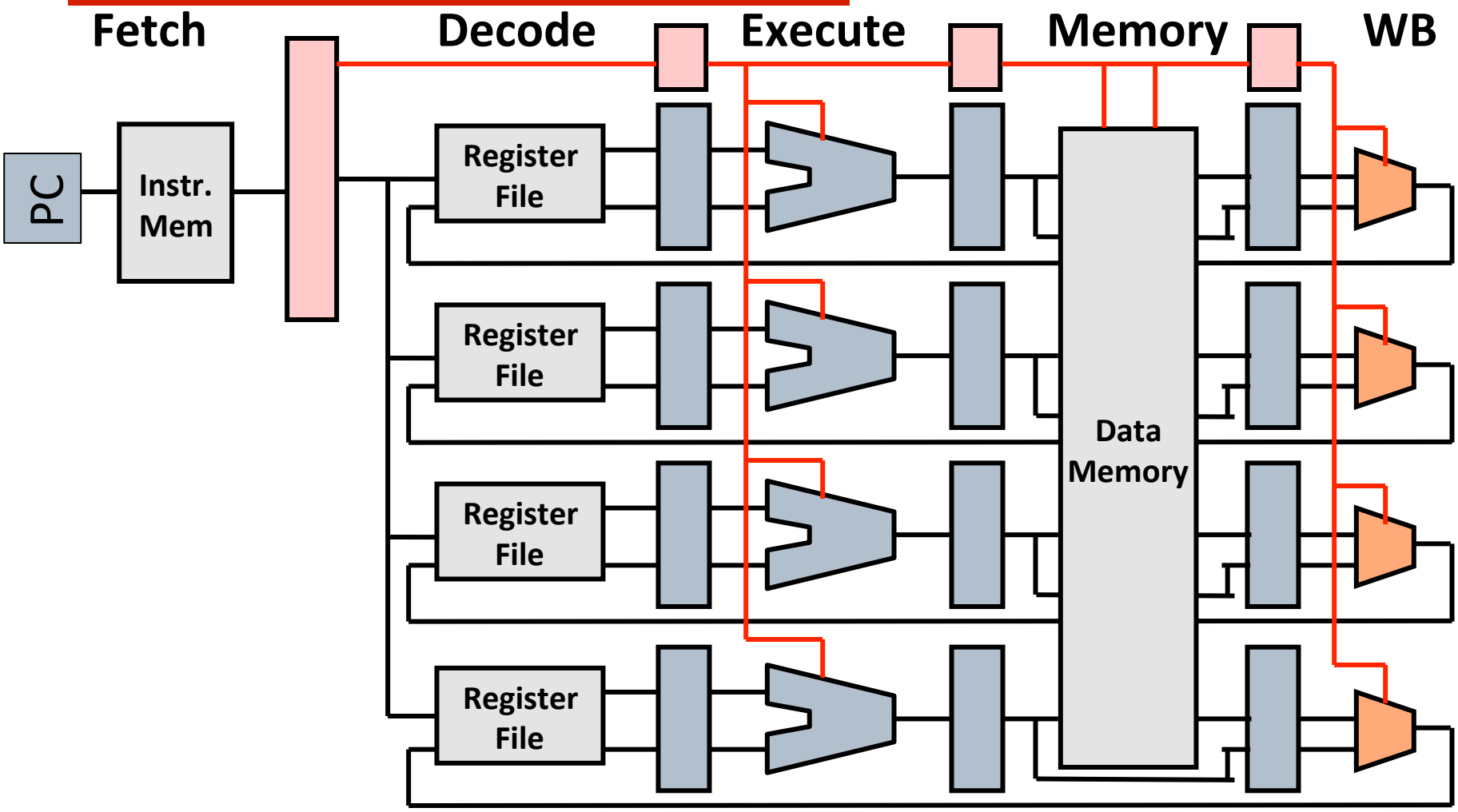


Other Techniques for TLP: Multi-Threading

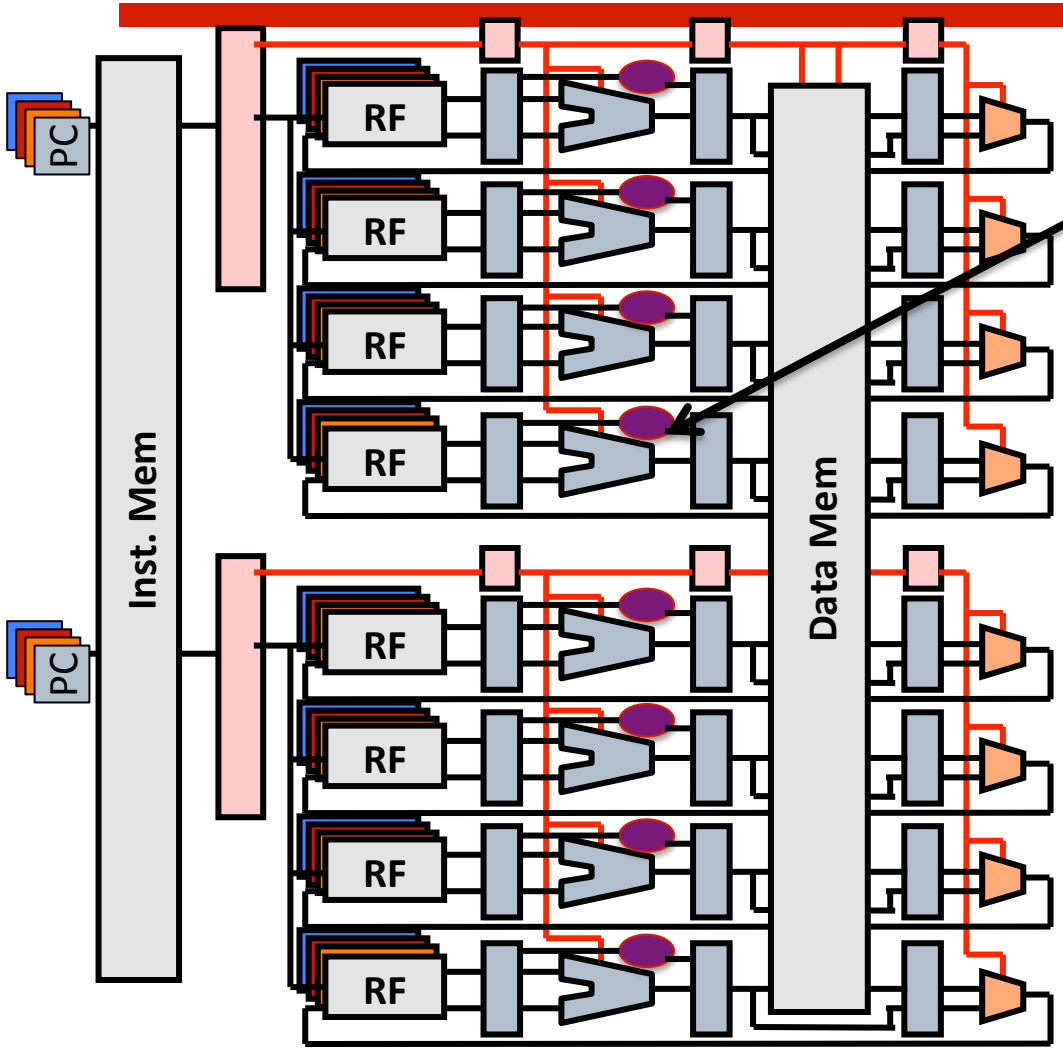


- Virtual Multiprocessor (Multi-Threading or HyperThreading)
  - Duplicate the state (PC, Registers) but time share hardware
  - User/Operating system see 2 cores, but only one execution
  - Used to hide long latencies (i.e. memory access to disk)

DLP Techniques: Single Instr. Multiple Data (SIMD)



Building a GPU



- Add special functional units in EX
- Combine Techniques
  - SIMD + MP + MT= SIMT
- MT used to hide memory latencies
- SIMD used to decrease power of fetch/control
- MP used to improve throughput