

# Cache Design 2

CS 1541

Wonsun Ahn

# Cache Design Parameter 6: Write-Through vs. Write-Back

---

# Writes and Cache Consistency

- Assume **&x** is  $111010_2$ , and  $x == 24$  initially

**lw** t0, &x

000

**addi** t0, t0, 1 # x++

001

**sw** t0, &x

010

- How will the **lw** change the cache?

011

- How will the **sw** change the cache?

100

- Uh oh, now the cache is **inconsistent**.  
(Memory still has the old value 24.)

101

110

111

V	Tag	Data
0		
0		
1	111	25
0		
0		
0		
0		
0		

- How can we solve this? Two policies:

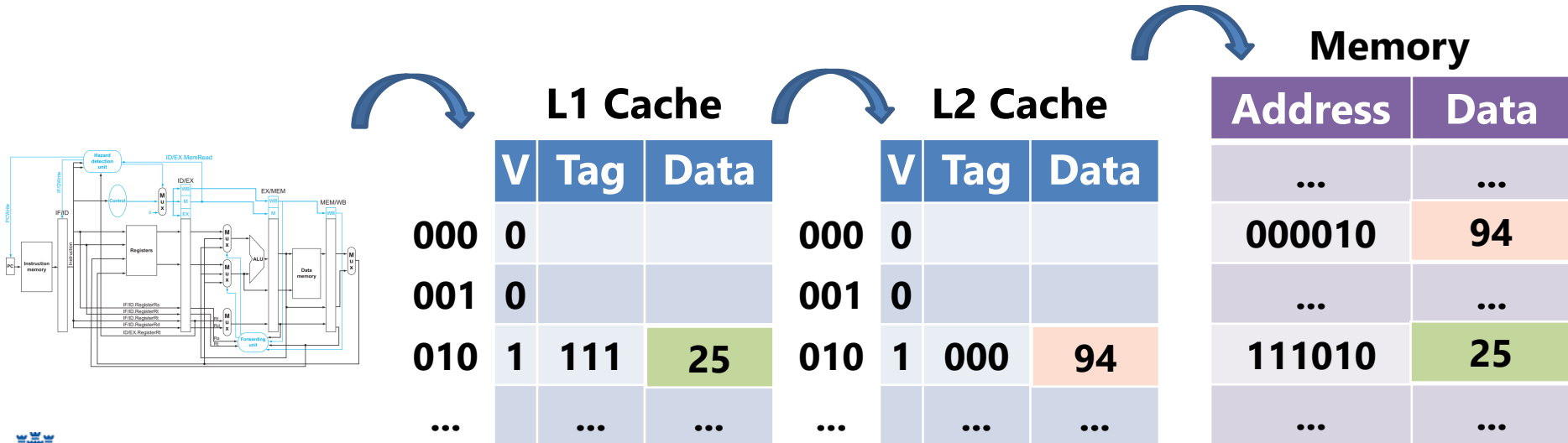
- Write-through**: Propagate write all the way through memory
- Write-back**: Write back cache block when it is evicted from cache

# Write-Through Policy

---

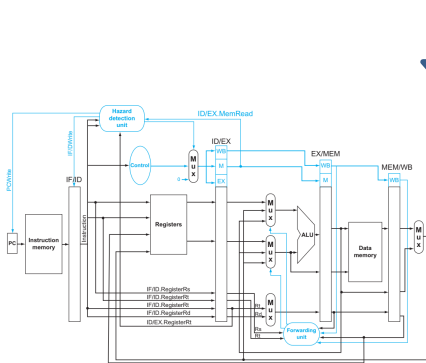
# Policy 1: Write-through

- Write-through:
    - On hit, write to cache block and **propagate** write to lower memory
    - On miss, keep on propagating the write to lower memory
  - What happens if we write **25** to address **111010<sub>2</sub>**?
  - What happens if we write **94** to address **000010<sub>2</sub>**?
- Caches are kept consistent at all points in time!



# Write-through: Allocate reads but not writes

- What happens if we read from address **000010<sub>2</sub>**?
  - We can just discard the conflicting cache block **111010<sub>2</sub>**
  - It's just an extra copy of the same data
- Note how we allocate blocks only on read misses
  - Write misses don't allocate blocks because it doesn't help anyway  
--- writes are propagated to lower memory even on write hits
  - This policy is called **no write allocate**

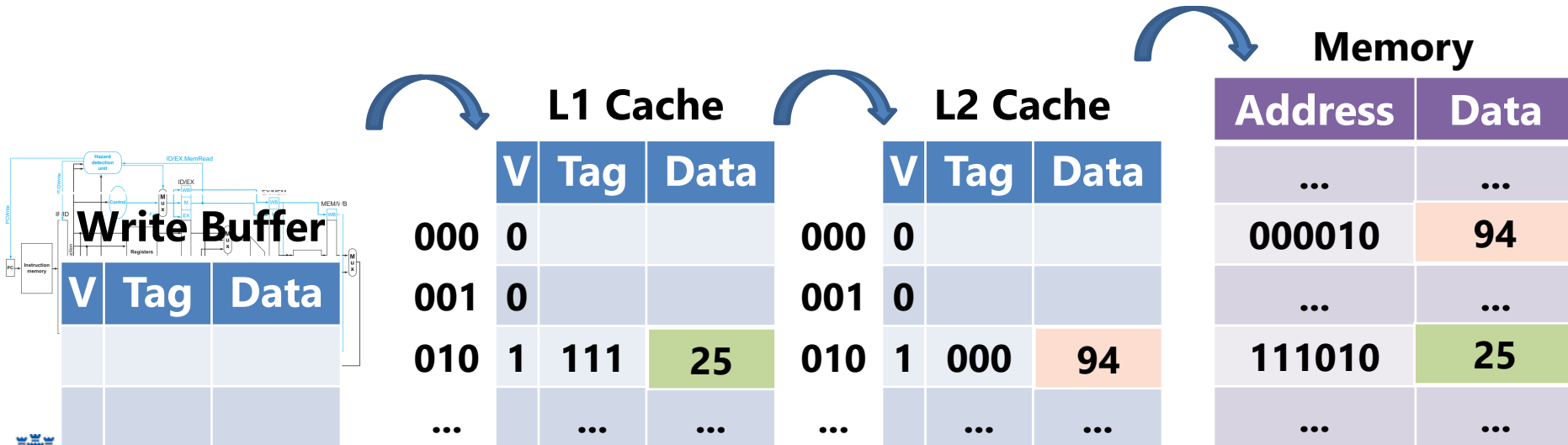


	L1 Cache				L2 Cache		
	V	Tag	Data		V	Tag	Data
000	0			000	0		
001	0			001	0		
010	1	000	94	010	1	000	94
...		...	...	...		...	...

Memory	
Address	Data
...	...
000010	94
...	...
111010	25
...	...

# Write-through: Drawbacks

- Drawback: **Long write delays** regardless of hit or miss
  - Must always propagate writes all the way to DRAM
- Solution: **Write buffer** maintaining pending writes
  - CPU gets on with work after moving pending write to write buffer
  - But does the write buffer solve all problems?



# Write-through: Drawbacks

- The write buffer does not solve all problems.
  1. Write buffer must be big enough to store all pending writes
    - May take more than 100 cycles for write to propagate to memory
    - Write buffer always checked on every load → may add to **hit time**
  2. Write buffer does not solve **bandwidth** problems
    - If memory bandwidth < rate of writes in program,
- Impractical to write-through all the way to memory
  - Typically only L1 caches are write-through, if any
- We need another strategy that is not so bandwidth-intensive

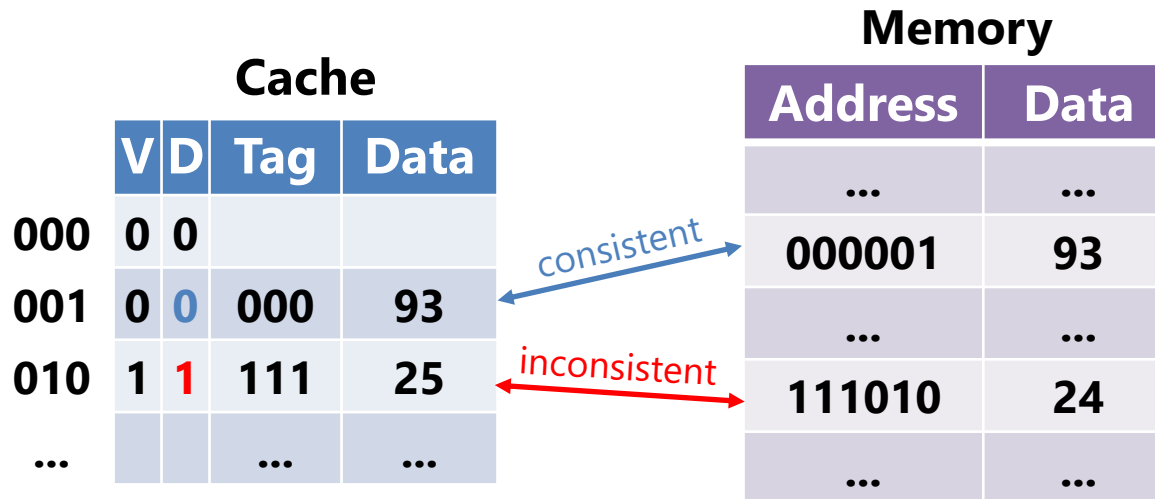


# Write-Back Policy

---

# Policy 2: Write-back

- **Dirty** block: a block that is temporarily inconsistent with memory
  - On a hit, write to cache block, marking it dirty. **No propagation.**
  - **Write back** dirty block to lower memory only when it is **evicted**  
→ Saves bandwidth since write hits no longer access memory
- A **dirty bit** is added to the cache block metadata (marked "D")
  - Block **000001**<sub>2</sub> is clean → can be discarded on eviction
  - Block **111010**<sub>2</sub> is dirty → needs to be written back on eviction



# Write-back: Allocate both reads and writes

- Unlike write-through, write hits do not result in propagation
  - May be beneficial to allocate writes to stop further misses!
  - Unlike write-through, write-back has a **write allocate** policy
- On write allocation, the block is **read** in from **lower memory**
  - Why the wasted effort?
  - Aren't we going to overwrite the block anyway with new data?
  - Why read in data that is going to be overwritten?

# Write-back: Write allocate

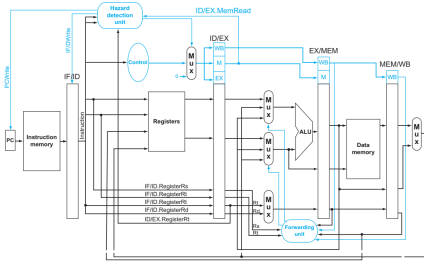
- Because a block is multiple bytes, and you are updating just a few
  - Suppose a cache block is 8 bytes (2 words)
  - Suppose you are writing to only the **first word**

V	D	Tag	Data	
1	1		<b>first word</b> (written)	<b>second word</b> (not written)

- After allocate, the entire cache block is marked **valid**
  - That means **second word** as well as **first word** must be valid
  - That means **second word** must be fetched from lower memory
  - Otherwise if later **second word** is read, it will contain junk data
  - Unavoidable, unless you have a valid bit for each byte
    - That means spending 1 bit for every 8 bits of data
    - That's just too much metadata overhead

# Write-back: Writes

- What happens if we write **25** to address **111010<sub>2</sub>**?



**Write Buffer**

V	Tag	Data

000

V	D	Tag	Data
0	0		
0	0		
1	0	111	24
...		...	...

000

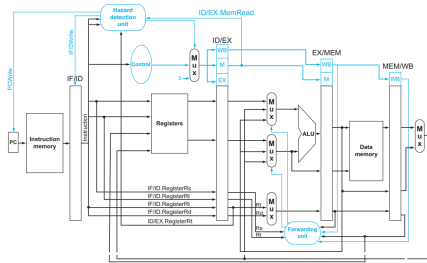
V	D	Tag	Data
0	0		
0	0		
1	0	000	93
...		...	...

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **25** to address **111010<sub>2</sub>**?
  - L1 Cache **hit!** Update cache block and mark it dirty.
  - That's it! How quick is that compared to write-through?



**Write Buffer**

V	Tag	Data

000  
001  
010  
...

V	D	Tag	Data
0	0		
0	0		
1	1	111	25
...		...	...

**L1 Cache**

000  
001  
010  
...

V	D	Tag	Data
0	0		
0	0		
1	0	000	93
...		...	...

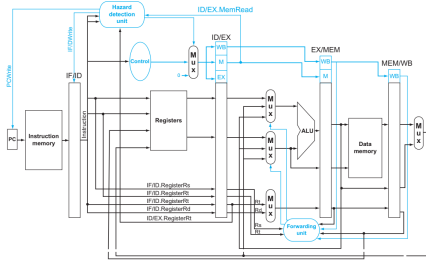
**L2 Cache**

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**?
  - L1 Cache **miss**! First thing we will do is add store to **Write Buffer**.  
(So that the CPU can continue executing past the store)



## Write Buffer

V	Tag	Data
1		94

000  
001  
010  
...

V	D	Tag	Data
0	0		
0	0		
1	1	111	25
...	...	...	...

## L1 Cache

000  
001  
010  
...

V	D	Tag	Data
0	0		
0	0		
1	0	000	93
...	...	...	...

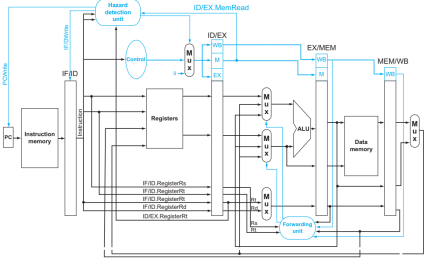
## L2 Cache

## Memory

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**? (cont'd)
  - Next the L2 Cache is searched and it's a **hit**!
  - To bring in block to L1 Cache, we first need to evict block **25**.
  - It's a dirty block, so we can't just discard it. Need to **write** it **back**!
  - Since block **25** misses in L2, it will take the **long** trip to Memory
  - Is there a way to put it aside and get to it later?



## Write Buffer

V	Tag	Data
1		94

000  
001  
010  
...

## L1 Cache

V	D	Tag	Data
0	0		
0	0		
1	1	111	25
		...	...



## L2 Cache

	V	D	Tag	Data
000	0	0		
001	0	0		
010	1	0	000	93
...			...	...

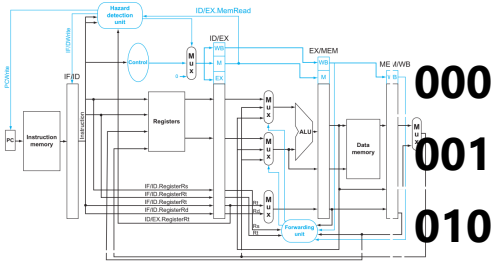
# Memory

Address	Data
...	...
000010	93
...	...
111010	24
...	...



# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**? (cont'd)
  - Yes! Add Write Buffers to caches, just like we did for the pipeline!
  - Move block to L1 Write Buffer so L1 Cache can continue working
  - Pending block will get written back to Memory eventually



**Write Buffer**

V	Tag	Data
1		94

**L1 Cache**

V	D	Tag	Data
0	0		
0	0		
1	1	111	25

**Write Buffer**

V	D	Tag	Data
0	0		
0	0		

**L2 Cache**

V	D	Tag	Data
0	0		
0	0		
1	0	000	93

**Write Buffer**

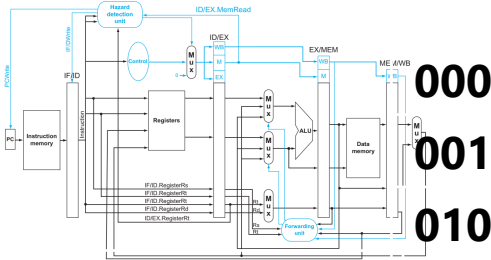
V	D	Tag	Data
0	0		
0	0		

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**? (cont'd)
  - Now we can finally read in block **93** to the L1 Cache



**Write Buffer**

V	Tag	Data
1		94

**L1 Cache**

V	D	Tag	Data
0	0		
0	0		
0	0		

**Write Buffer**

V	D	Tag	Data
1	1	111	25
0	0		

**L2 Cache**

V	D	Tag	Data
0	0		
0	0		
1	0	000	93

**Write Buffer**

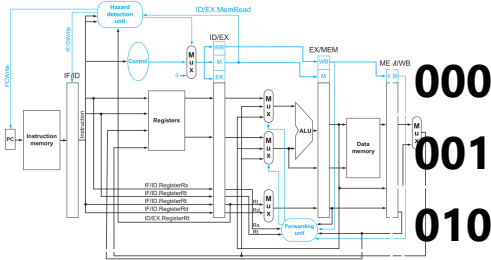
V	D	Tag	Data
0	0		
0	0		

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**? (cont'd)
  - Now we can finally read in block **93** to the L1 Cache
  - And write **94** into the cache block, also marking it dirty
  - Store is finished, so now remove it from pipeline Write Buffer!



**Write Buffer**

V	Tag	Data
1		94

**L1 Cache**

V	D	Tag	Data
0	0		
0	0		
1	1	000	94

**Write Buffer**

V	D	Tag	Data
1	1	111	25
0	0		

**L2 Cache**

V	D	Tag	Data
0	0		
0	0		
1	0	000	93

**Write Buffer**

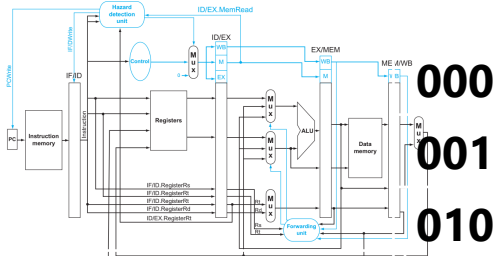
V	D	Tag	Data
0	0		
0	0		

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

# Write-back: Writes

- What happens if we write **94** to address **000010<sub>2</sub>**? (cont'd)
  - Eventually, the pending block in L1 Write Buffer will write back
  - But this didn't affect the original store latency



**Write Buffer**

V	Tag	Data

**L1 Cache**

V	D	Tag	Data
0	0		
0	0		
1	1	000	94

**Write Buffer**

V	D	Tag	Data
1	1	111	25
0	0		

**L2 Cache**

V	D	Tag	Data
0	0		
0	0		
1	0	000	93

**Write Buffer**

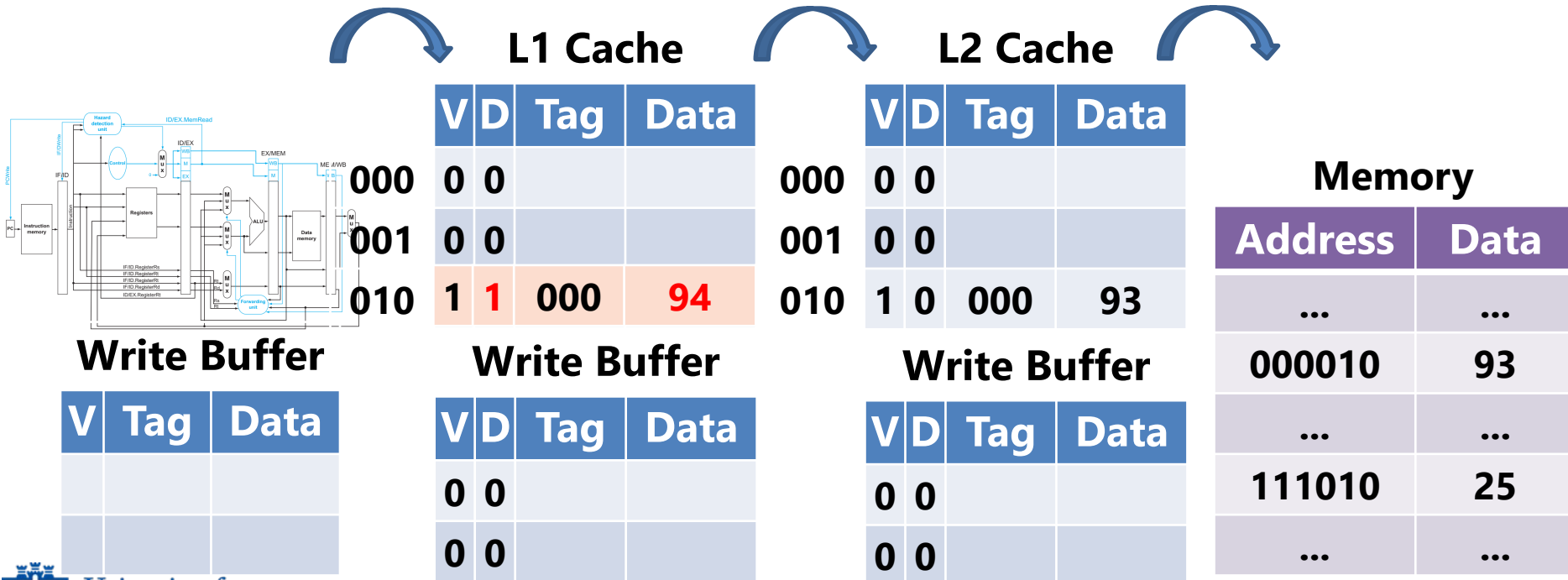
V	D	Tag	Data
0	0		
0	0		

**Memory**

Address	Data
...	...
000010	93
...	...
111010	24
...	...

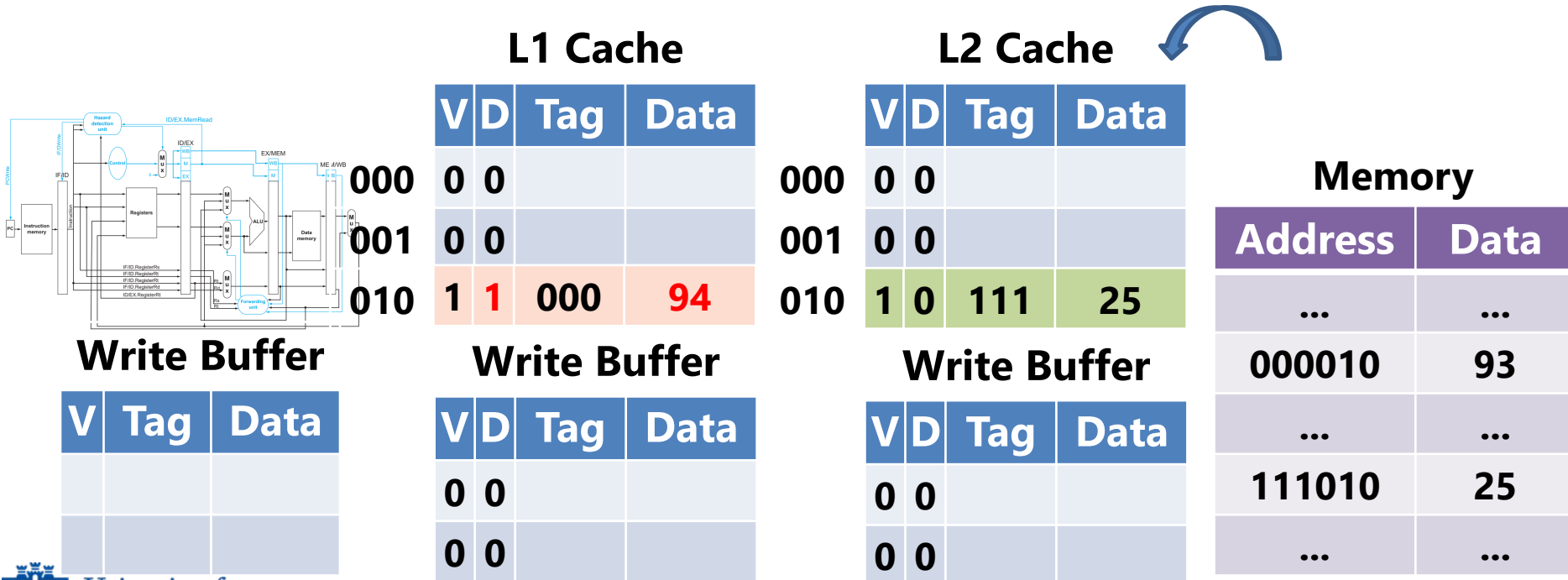
# Write-back: Reads

- What happens if we read **25** from address **111010<sub>2</sub>**?
  - Misses in L1 and L2 caches and must go all the way to Memory



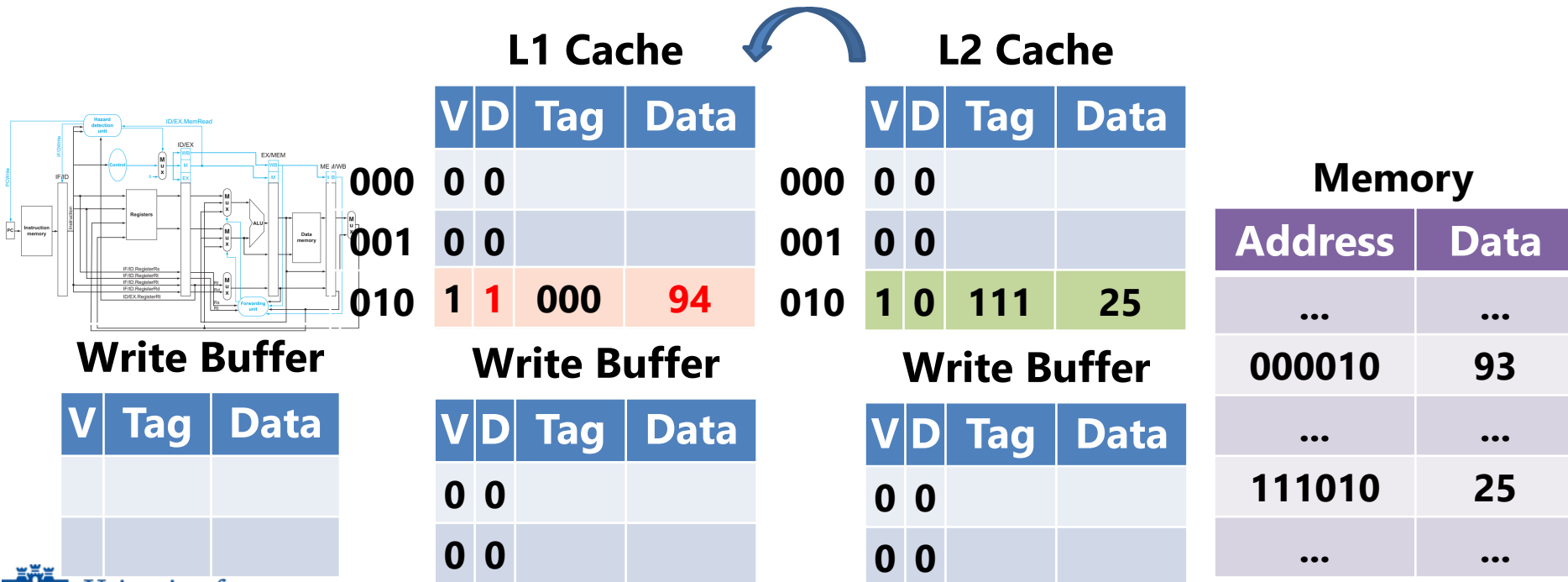
# Write-back: Reads

- What happens if we read **25** from address **111010<sub>2</sub>**?
  - Misses in L1 and L2 caches and must go all the way to Memory
  - Fills the L2 Cache with **25** on the way back after evicting block **93**  
(Note that block **93** can simply be discarded since it's clean)



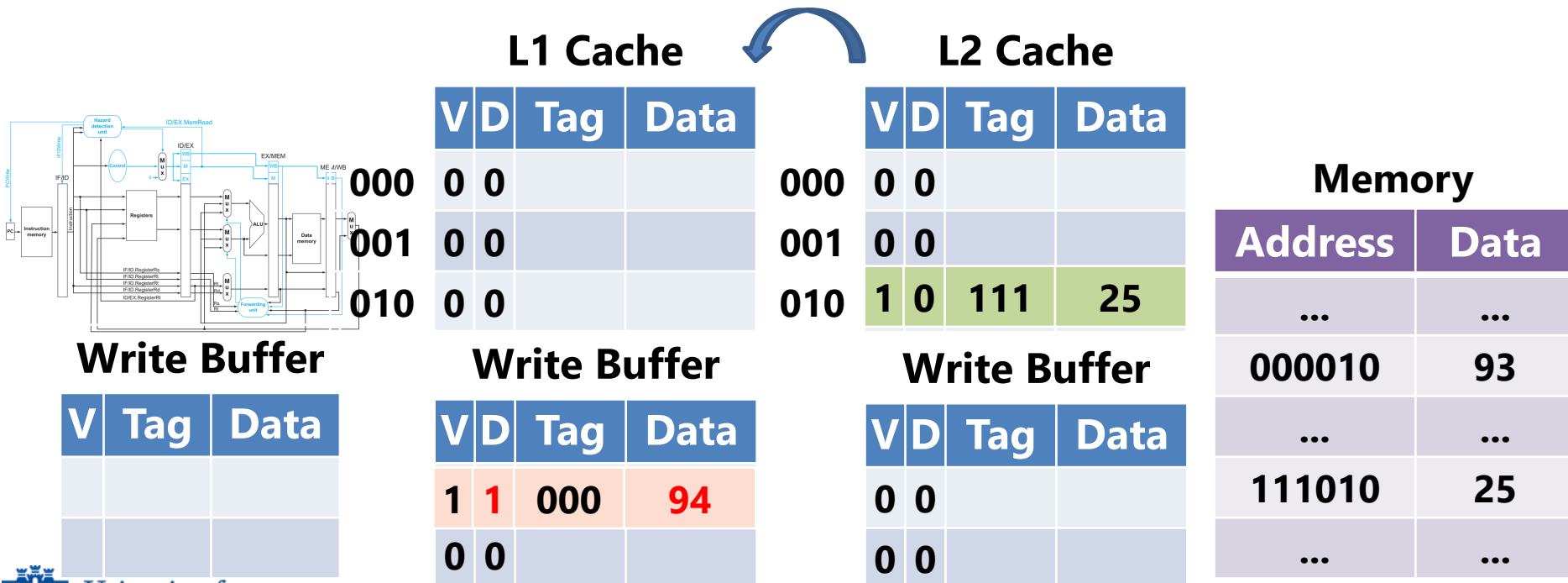
# Write-back: Reads

- What happens if we read **25** from address **111010<sub>2</sub>**? (cont'd)
  - Now it needs to evict block **94** in L1 Cache before filling with **25**
  - But block **94** needs to be written back since it's dirty!
  - So move to Write Buffer temporarily to make space.



# Write-back: Reads

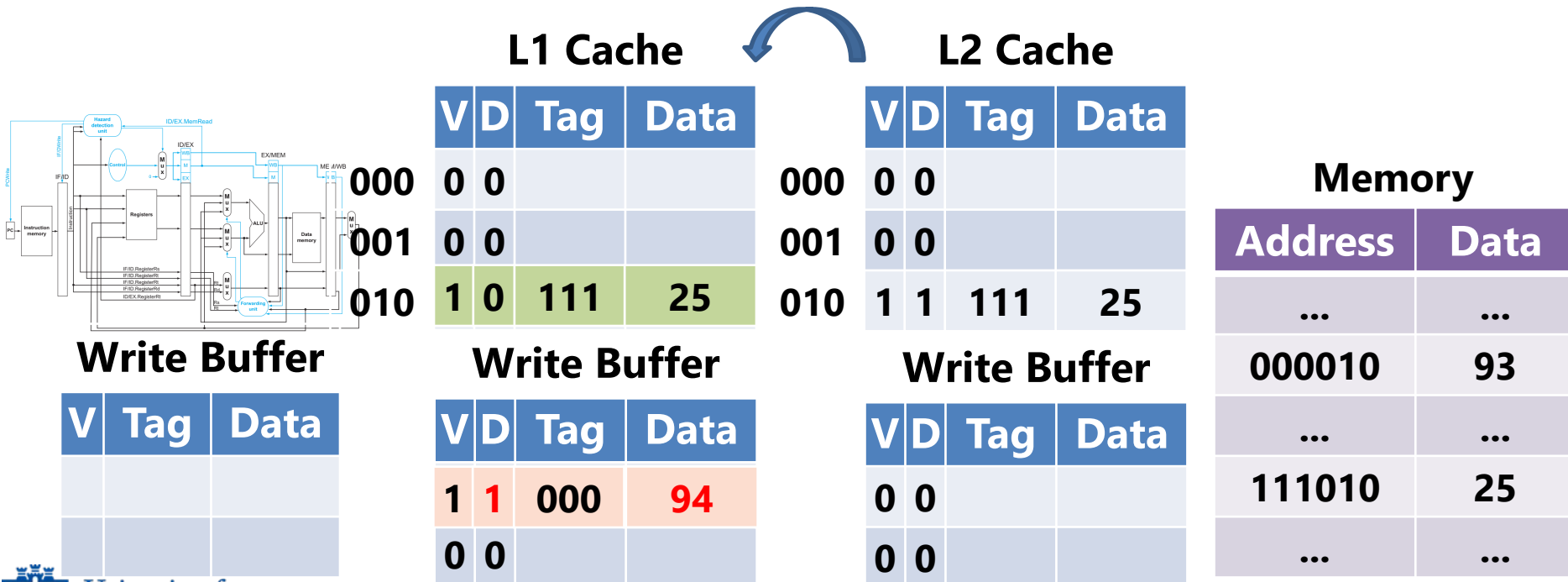
- What happens if we read **25** from address **111010<sub>2</sub>**? (cont'd)
  - Now L1 Cache can be filled with block 25





# Write-back: Reads

- What happens if we read **25** from address **111010<sub>2</sub>**? (cont'd)
  - Now L1 Cache can be filled with block **25**
  - Block **94** will eventually be written back to Memory
  - Write buffers in this context are also called **victim caches**



# Impact of Write Policy on AMAT

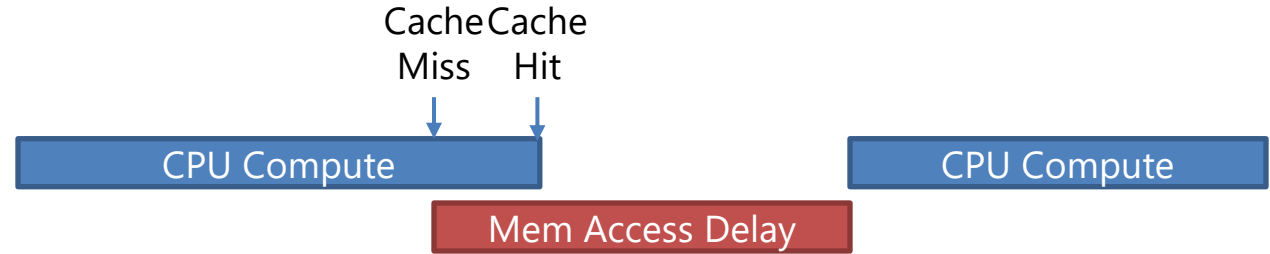
- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- **Write-through** caches have a **longer write hit time**
  - With write-back, a read hit and write hit take the same amount of time
  - With write-through, a write hit takes the same time as a write miss
- **Write-back** caches can have **longer miss penalty**
  - Due to write-backs of dirty blocks when making space for new block
- Both issues can be mitigated using **write buffers** to varying degrees
- All in all, write-back caches usually outperform write-through caches
  - Because write hits are typically more frequent compared to write-backs
- But write-through sometimes used in L1 cache due to simplicity
  - Plenty of L1  $\rightarrow$  L2 (intra-chip) bandwidth to handle write propagation
  - For L3, L3  $\rightarrow$  DRAM bandwidth cannot support write propagation

# Cache Design Parameter 7: Blocking vs. Non-blocking

---

# Blocking caches can only service one request at a time

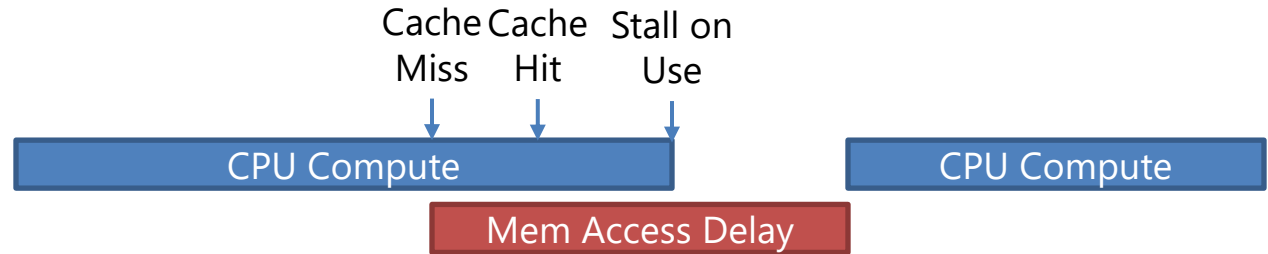
- Blocking caches:



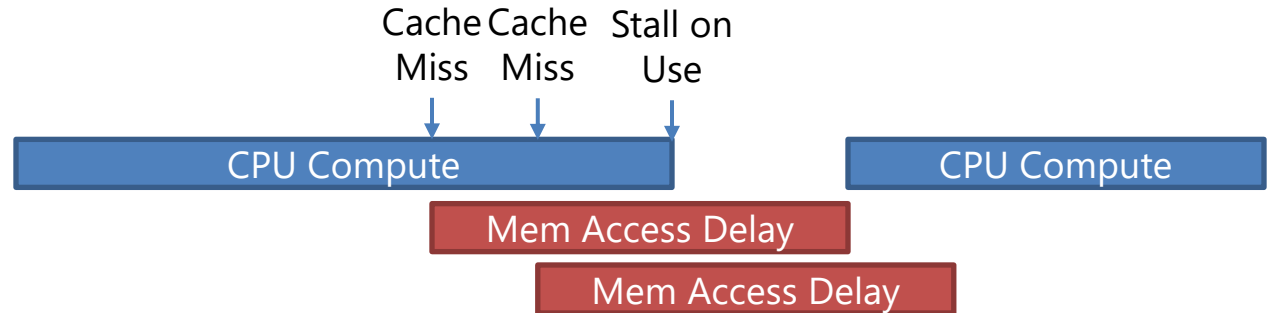
- After cache miss, cache goes into “not ready” state until it returns
- CPU can continue computation through out-of-order execution
- But, on second cache access, cache will not respond even on hit

# Non-blocking caches service requests concurrently

- Hit under miss:

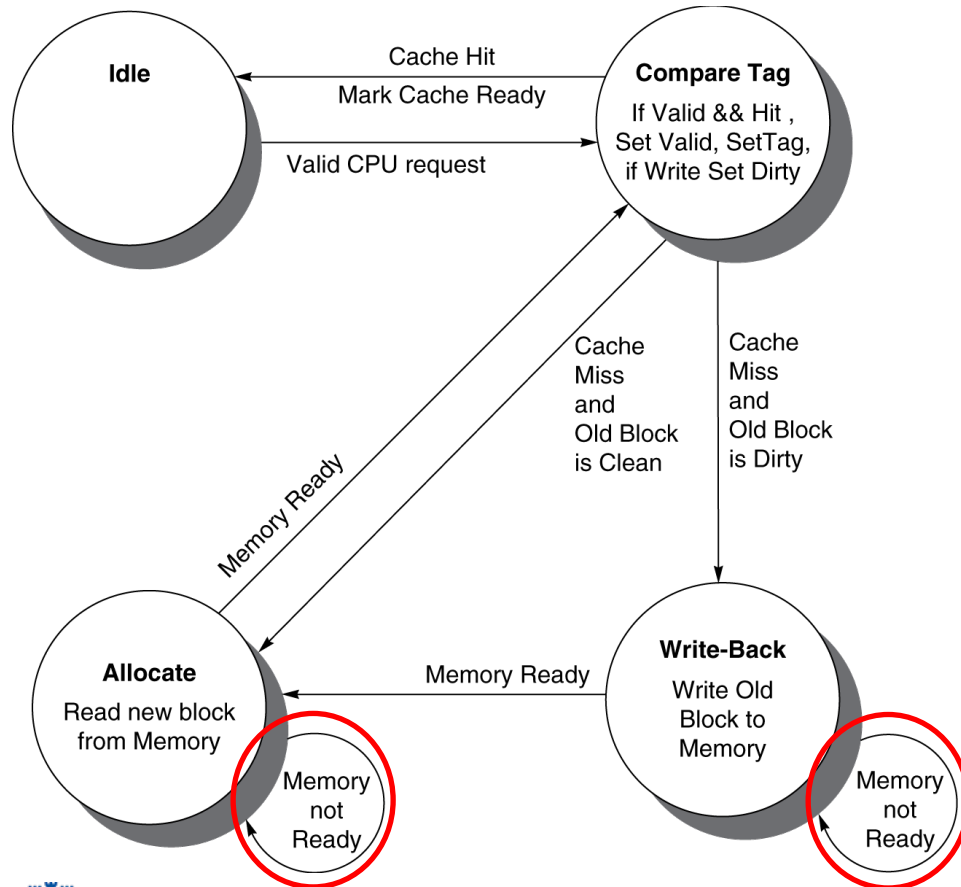


- Miss under miss:



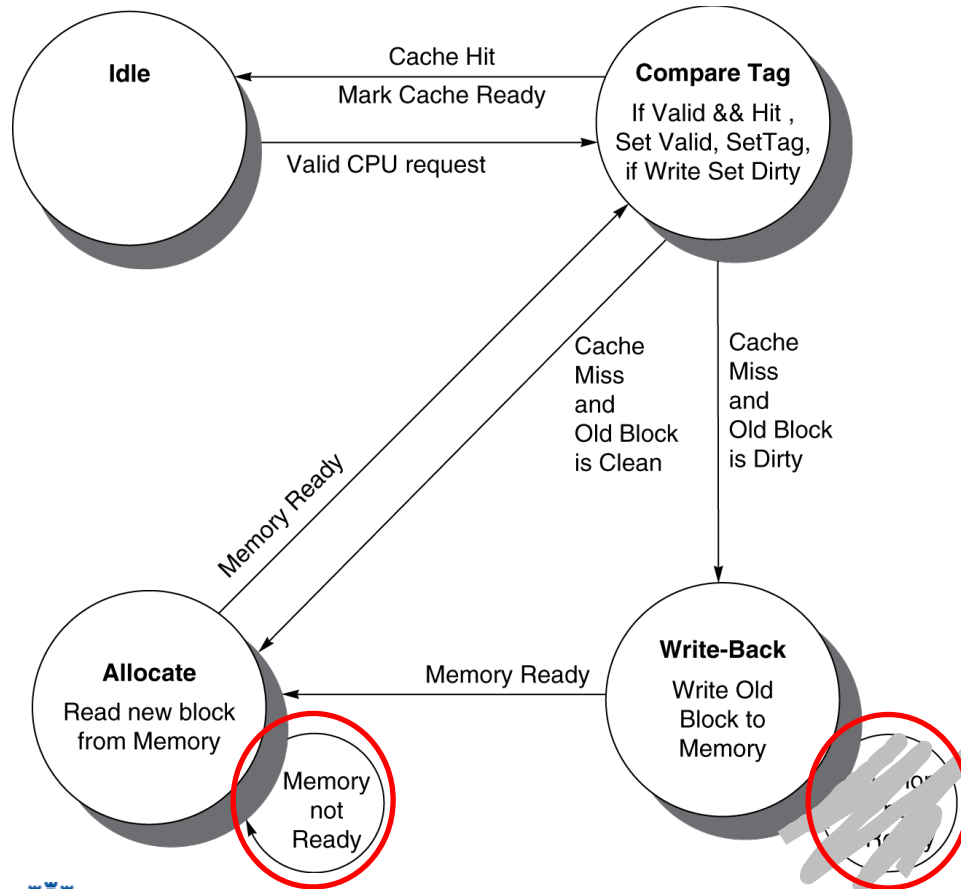
- **Non-blocking cache** allows both to happen
  - Allows **Memory Level Parallelism (MLP)**
  - As important to performance as Instruction Level Parallelism (ILP)

# Blocking Cache FSM for Write Back Caches



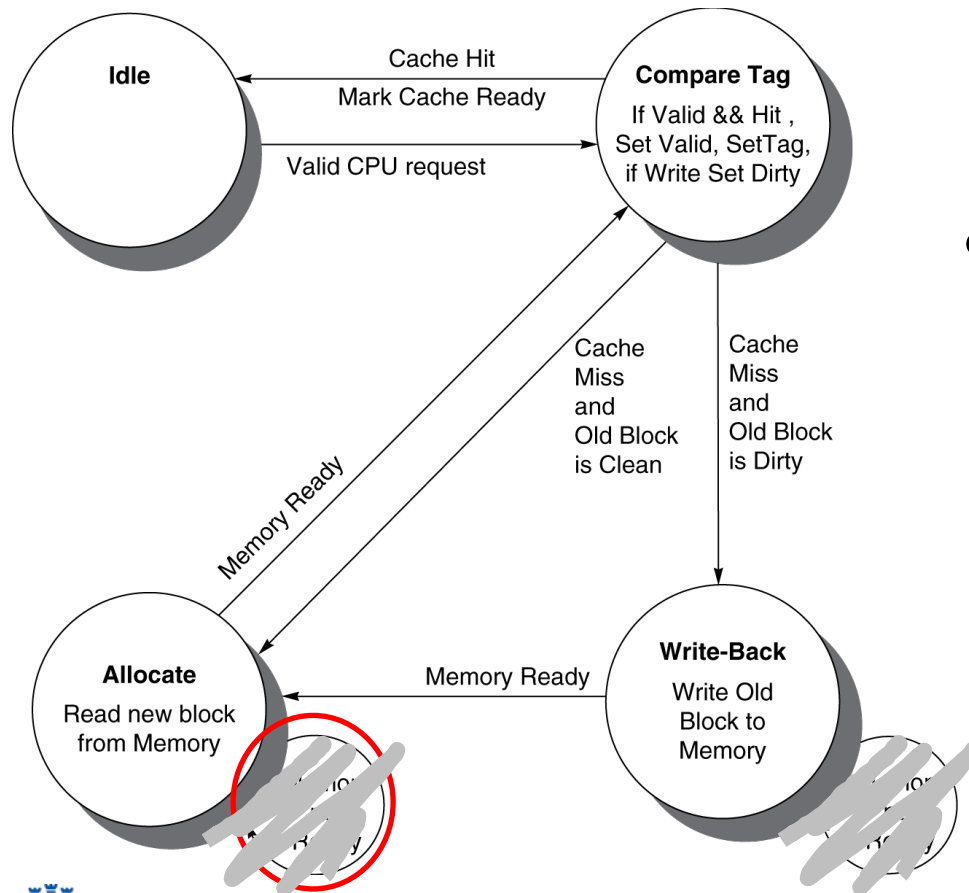
- FSM must be in **Idle** state for cache to receive new requests
- Prolonged "Memory not Ready" blocks quick return to idle state → Called **Blocking Cache**

# Non-Blocking Cache FSM w/ Write Buffer



- FSM must be in **Idle** state for cache to receive new requests
- Prolonged "Memory not Ready" blocks quick return to idle state → Called **Blocking Cache**
- **Write buffer** allows cache to defer **write-back** until later
  - Quick transition to allocate
- But how about "Memory not Ready" on Allocate state?

# Non-Blocking Cache FSM w/ Write Buffer + MSHR



- On cache miss:
  - Miss registered in MSHR.
  - Cache quickly returns to idle.
- **Miss Status Handling Register**
  - A table that keeps track of pending missed requests.
  - When request returns,
    - Cache block filled with data.
    - Request forwarded up the memory hierarchy.



# Impact of non-blocking caches

- Non-blocking caches do not impact our three cache metrics
  - Hit time, miss rate, and miss penalty remain mostly the same
- Impact is that **miss penalty** can be **overlapped** with:
  - Computation of instructions not dependent on the miss
  - Miss penalties of other memory requests
- Out-of-order processors are always coupled with a non-blocking cache
  - Otherwise, the ability to do out-of-order execution is severely stymied

# Cache Design Parameter 8: Unified vs. Split

---

# Problem with Split Caches

- If cache is split into two (i-cache and d-cache)
  - Space cannot be flexibly allocated between data and code



If our working set looks like this – say, in a small loop traversing a large array – then we run out of data space.



If our working set looks like this – say, in a large loop that uses little data – then we run out of code space.

# Impact of Unifying Cache

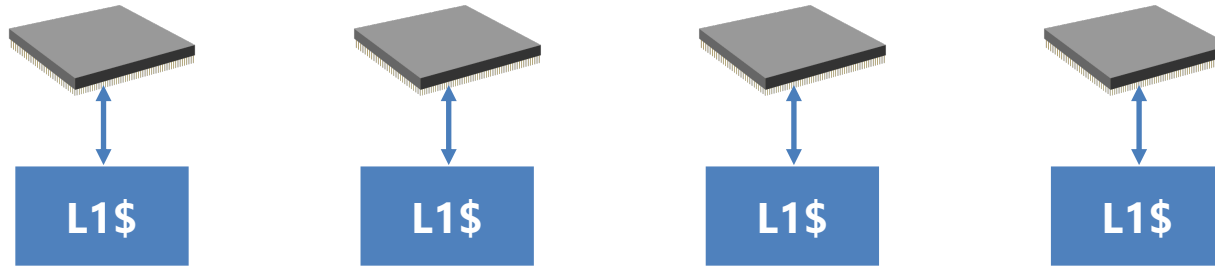
- The answer to the problem is to simply **unify** the cache into one
- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Impact of unifying cache on miss rate:
  - **Smaller miss rate** due to more flexible use of space
- Impact of unifying cache on hit time:
  - Potentially **longer hit time** due to structural hazard
  - With split caches, i-cache and d-cache can be accessed simultaneously
  - With unified cache, access request must wait until port is available
- **L1** cache is almost always **split**
  - Frequent accesses directly from pipeline trigger structural hazard often
- **Lower level** caches are almost always **unified**
  - Accesses are infrequent (filtered by L1), so structural hazards are rare

# Cache Design Parameter 9: Private vs. Shared

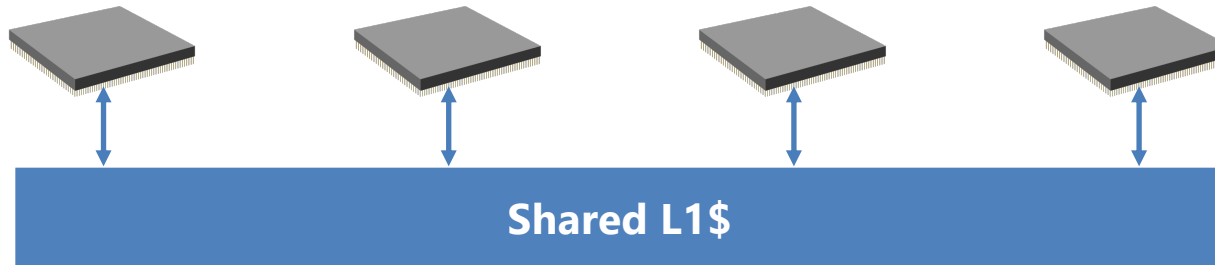
---

# Private vs. Shared Cache

- On a multi-core system, there are two ways to organize the cache
- **Private** caches: each core (processor) uses its own cache

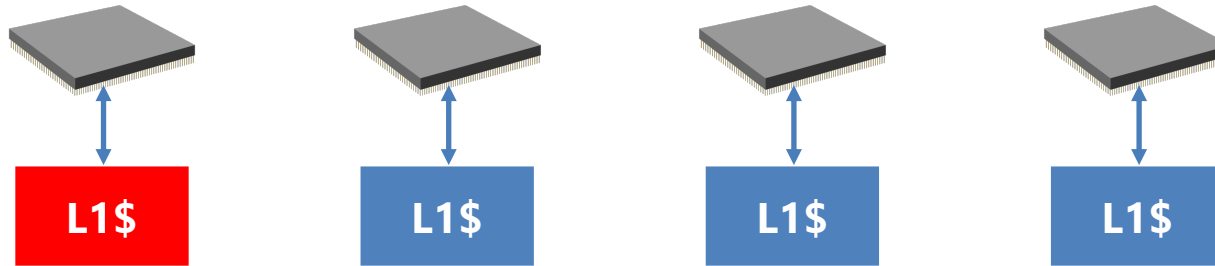


- **Shared** cache: all the cores share one big cache



# Shared Cache can Use Space More Flexibly

- Suppose only 1<sup>st</sup> core is active and other cores are idle
  - How much cache space is available to 1<sup>st</sup> core? (Shown in **red**)
- **Private** caches: 1<sup>st</sup> core can only use its own private cache

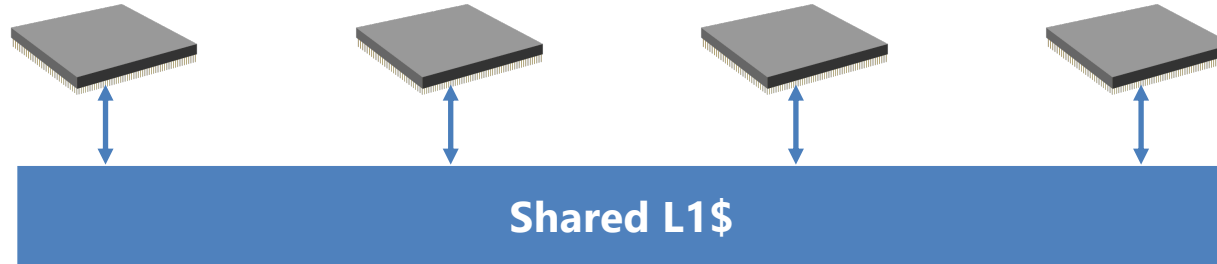


- **Shared** cache: 1<sup>st</sup> core can use entire shared cache!

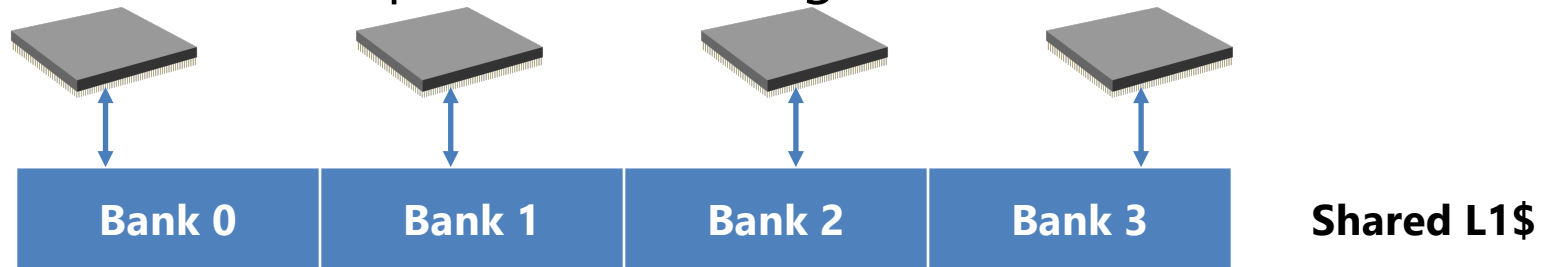


# Banking: Solution to Structural Hazards

- Now what if all the cores are active at the same time?
  - Won't that cause **structural hazards** due to simultaneous access?



- Could add more ports, but adding **banks** is more cost effective

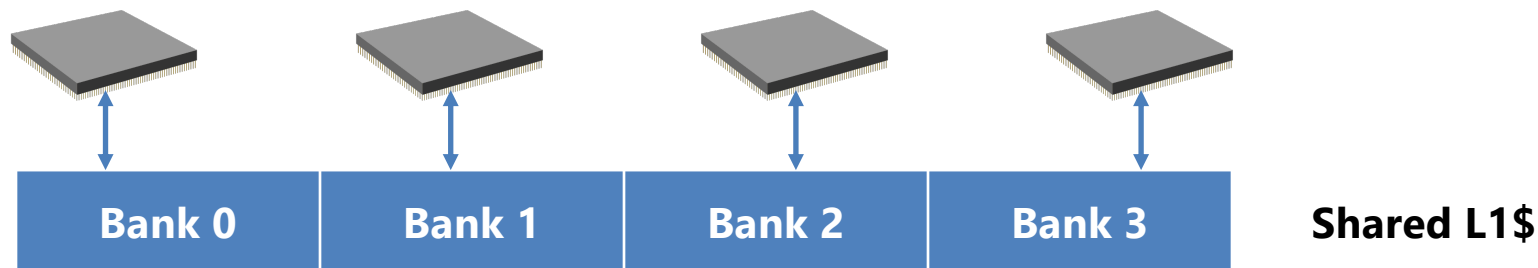


- Each bank has one read / write port
- As long as two cores do not access same bank, no hazard!



# Banking: Solution to Structural Hazards

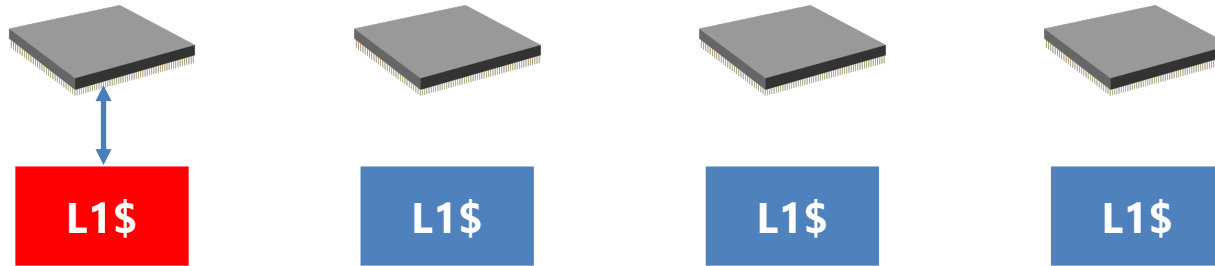
- Cache blocks are **interleaved** between banks



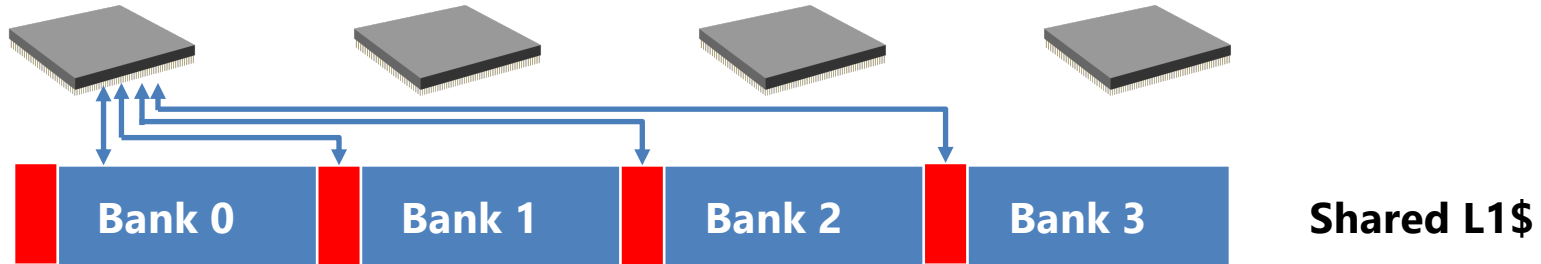
- Blocks 0, 4, 8 ... → Bank 0
- Blocks 1, 5, 9 ... → Bank 1
- Blocks 2, 6, 10 ... → Bank 2
- Blocks 3, 7, 11 ... → Bank 3
- Why?
  - Spatial locality means cores likely to work on adjacent memory
  - Causes cache accesses to be evenly distributed → less hazards

# Shared Cache have Longer Access Times

- Again, suppose only 1<sup>st</sup> core is active and other cores are idle
  - The working set data is shown in **red**
- **Private** caches: entire working set data in nearby private cache

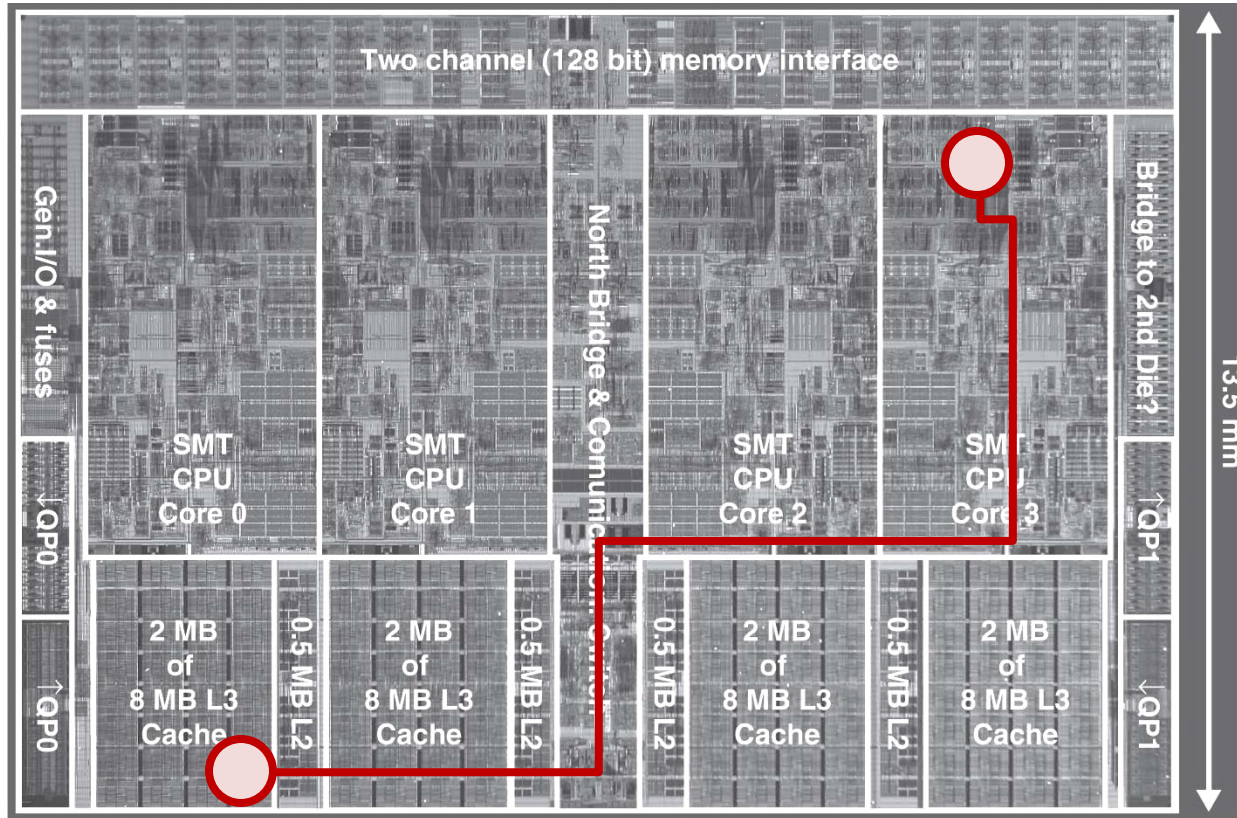


- **Shared** cache: data likely distributed to remote banks



# Shared Cache have Longer Access Times

- Remember this picture?

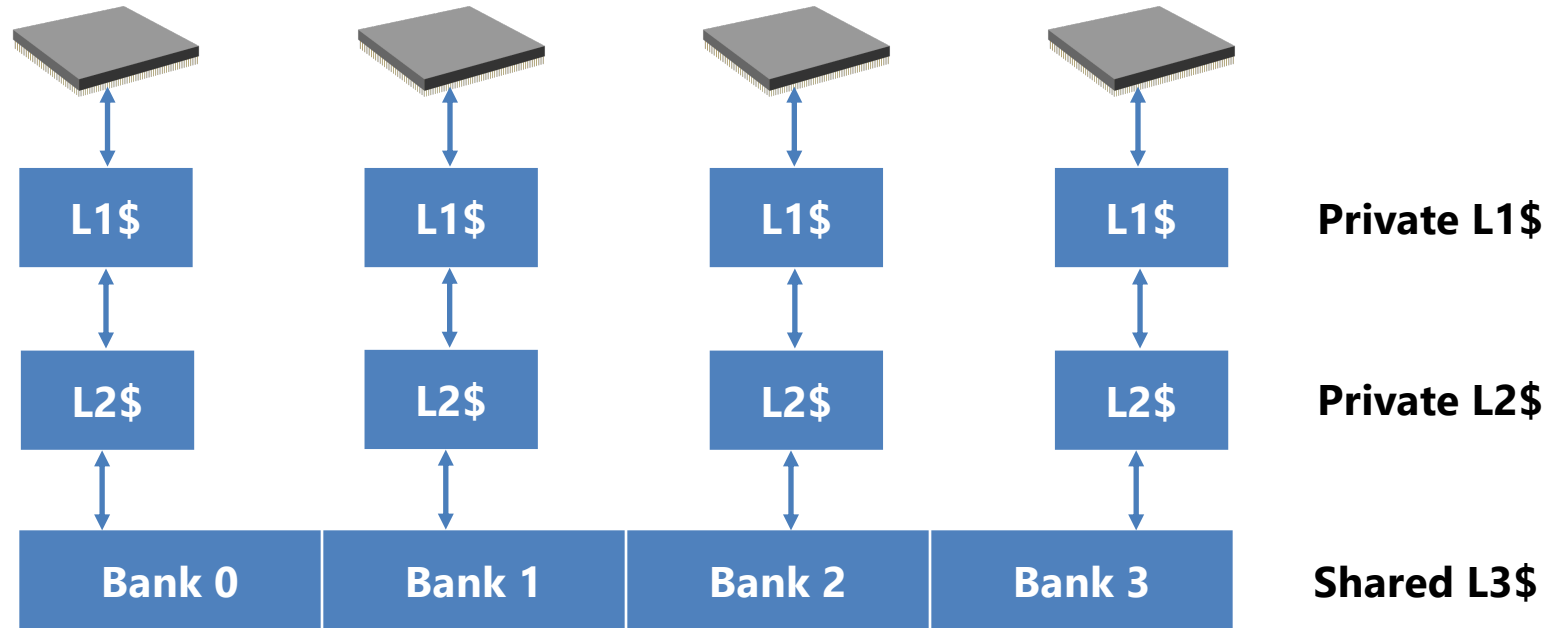


# Impact of Shared Cache

- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Impact of shared cache on miss rate:
  - **Smaller miss rate** due to more flexible use of space
- Impact of shared cache on hit time:
  - **Longer hit time** due to sometimes having to access remote banks
- **L1** caches are almost always **private**
  - Hit time is important for L1. Cannot afford access to remote banks.
- **L3 (last level)** caches are almost always **shared**
  - Reducing miss rate is top priority to avoid DRAM access.

# Cache Organization of Broadwell CPU

- This is the cache organization of Broadwell used in our Linux server



- Intel rebrands the shared cache as the "Smart Cache"

# Cache Design Parameter 10: Prefetching

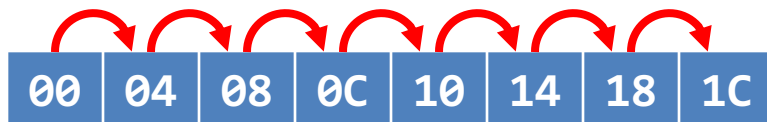
---

# Prefetching

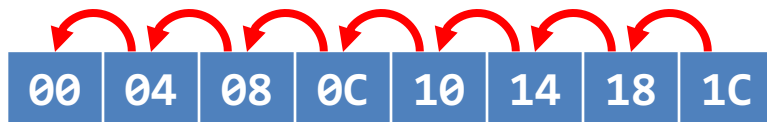
- **Prefetching**: fetching data that is expected to be needed soon
  - Allows you to hide the latency of fetching that data
  - E.g. Web browsers prefetch resources from not-yet-clicked links
    - when user later clicks on link, response is almost instantaneous
  - Caches also prefetch data that is expected to be used soon
    - Can be used to avoid even **cold misses**
- Two ways prefetching can happen:
  - Compiler-driven: compiler emits **prefetch instructions**
    - Can manually insert one in C program: `__builtin_prefetch(addr)`
    - Or rely on compiler to insert them using heuristics
  - Hardware-driven: CPU **prefetcher** emits prefetches dynamically
    - Relies on prefetcher to detect a pattern in memory accesses

# Hardware Prefetching

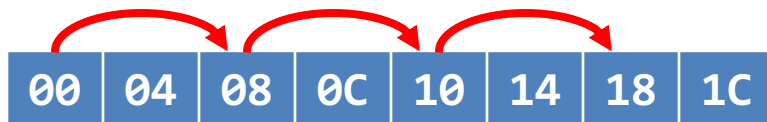
- What do you notice about both these snippets of code?
- They both access memory **sequentially**. **for**(i = 0 .. 100000) A[i]++;
- These kinds of access patterns are very common.



Sequential



Reverse sequential



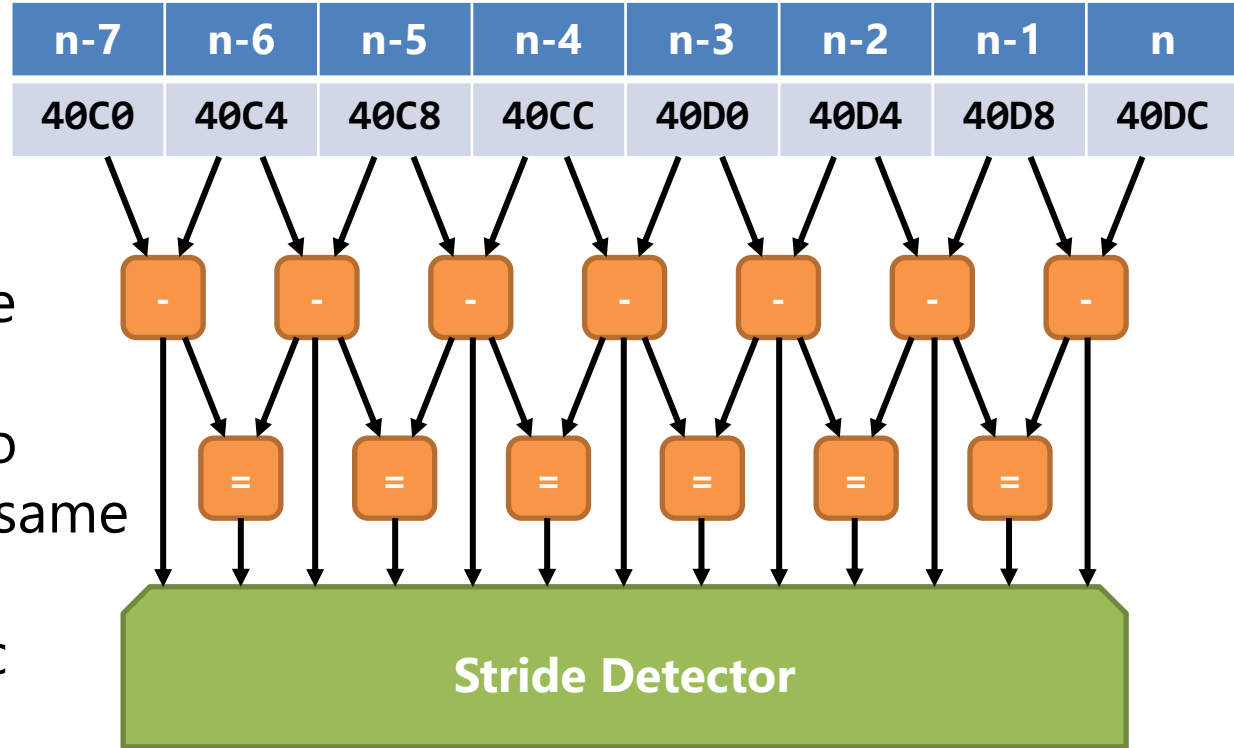
Strided sequential  
(think "accessing one field  
from each item in an array  
of structs")

00 lw  
04 lw  
08 lw  
0C addi  
10 sub  
14 mul  
18 sw  
1C sw  
20 sw



# Hardware Prefetching Stride Detection

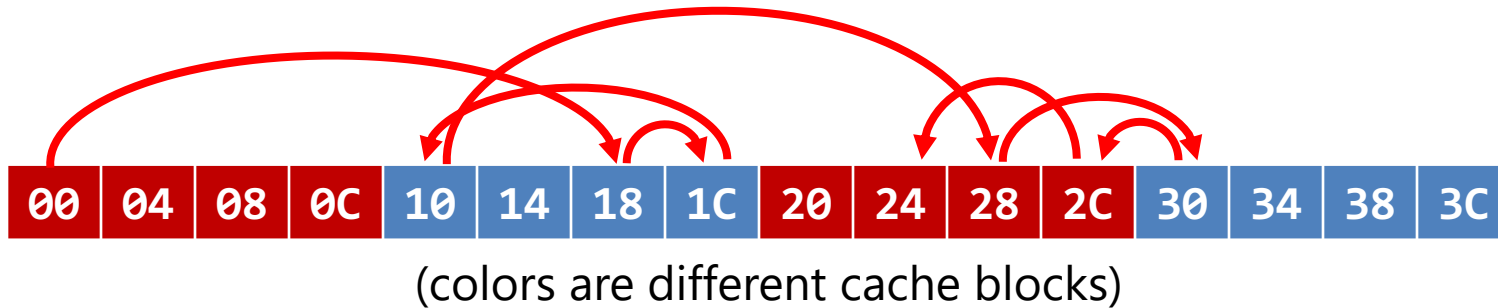
- What kinds of things would you need?
- A table of the last  $n$  memory accesses would be a good start.



- Some subtractors to calculate the stride
- Some comparators to see if strides are the same
- Some detection logic

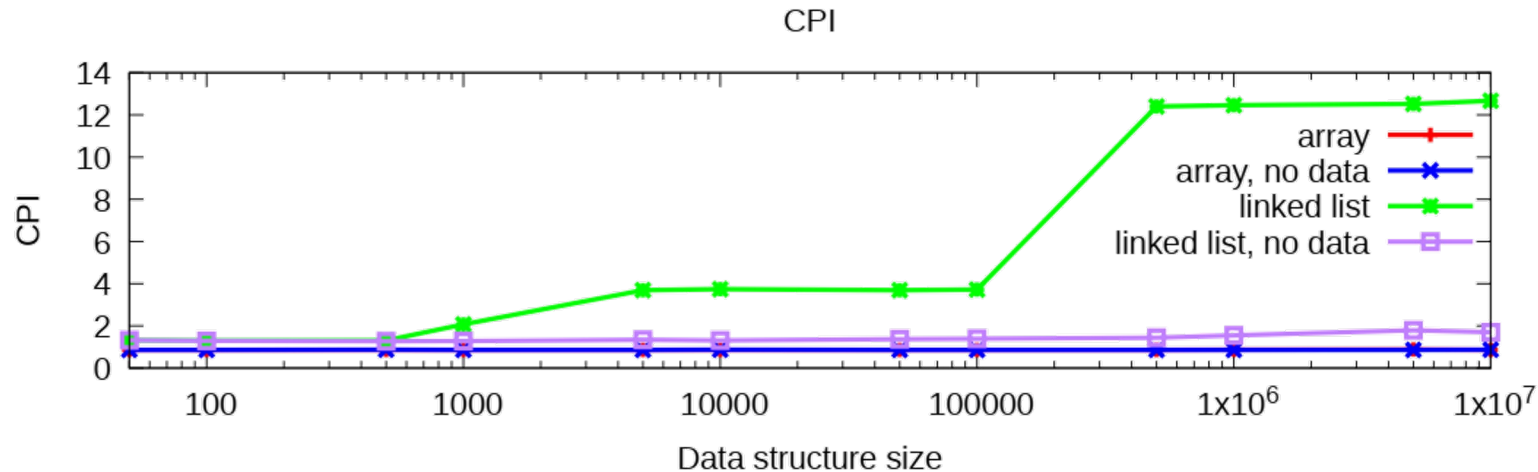
# Where Hardware Prefetching Doesn't Work

- **Sequential** accesses are where prefetcher works best
  - E.g. Iterating over elements of an **array**
- Some accesses don't have a pattern or is too **complex** to detect
  - At below is how a typical **linked-list** traversal looks like



- Other pointer-chasing data structures (**graphs**, **trees**) look similar
- Can only rely on naturally occurring locality to avoid misses
- Or, have compiler insert prefetch instructions in middle of traversal

# Mystery Solved



- How come **Array** performed well for even an array 1.28 GB large?
  - No spatial locality since each node takes up two 64-byte cache blocks
  - No temporal locality since working set of 1.28 GB exceeds any cache
- The answer is: **Array** had the benefit of a strided **prefetcher**!
  - Access pattern of **Linked List** was too complex for prefetcher to detect

# Impact of Prefetching

- Prefetcher runs **in parallel** with the rest of the cache hardware
  - Does not slow down any on-demand reads or writes
- What if prefetcher is wrong? It can be wrong in two ways:
  - It fetched a block that was never going to be used
  - It fetched a useful block but fetched it too soon or too late
    - Too soon: the block gets evicted before it can be used
    - Too late: the prefetch doesn't happen in time for the access
- A bad prefetch results in **cache pollution**
  - Unused data is fetched, potentially pushing out other useful data
- On the other hand, good prefetches can reduce misses drastically!