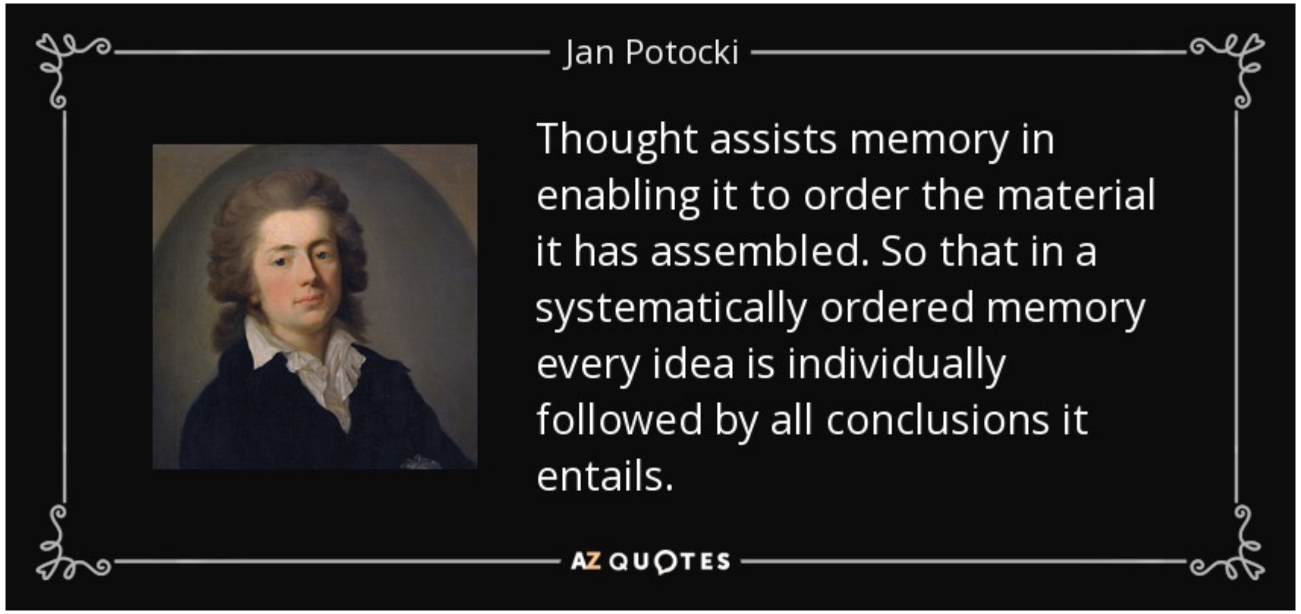
CS302

Memory Consistency I

Spring 2025
Arkaprava Basu & Babak Falsafi
parsa.epfl.ch/course-info/cs302



Jan Potocki

Thought assists memory in enabling it to order the material it has assembled. So that in a systematically ordered memory every idea is individually followed by all conclusions it entails.

AZ QUOTES

Adapted from slides originally developed by Prof. Falsafi
Copyright 2025

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
➡	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Memory Consistency
  - Ordering reads/writes
  - ISA level
- Exercise session
  - MPI demo continued
- Next Tuesday:
  - Taking consistency to the compiler and PL

FAQs on Ed

- Cannot diagnose the source of scalability bottleneck?
  - Time the different parts (e.g., for loops) of your program
  - All parallel parts should scale close to linearly with number of cores

- Optimizations for false sharing do not work?
  - Make sure padding is applied to the correct variables
  - Use performance counters to check if padding is working

- Recording problems for Week 3 Thursday:
  - Older version of lecture recording uploaded on Moodle
- Questions on Ed should be answered within a day (email us if urgent)

Assignment 1 and Assignment 2

- Assignment 1 deadline is this Sunday at 23:59!
  - Submit report and code in a single zip file on Moodle
  - No extensions!

- Assignment 2 to be released next Monday
  - Parallel programming using MPI

- If your partner has left/will leave the course, send us an email!
  - We will pair up these unpaired students for A2 and A3
  - Your partner must have officially dropped the course

Reminder: Hardware Cache Coherence

- Solves the problem of multiprocessors transparently sharing a single memory location
  - All processors agree on R/W order to address `X`
- Coherence makes caches appear invisible
  - Programmer can have the illusion of uniform memory with reduced latency due to the cache(s)
- But what about different memory locations?

Coherence vs. Consistency

- Memory **consistency** defines the behavior of R/W operations across different addresses
- Best illustrated with an example:
  - Assume `A & B` are addresses, r_x are registers

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

Cache Coherence Guarantees?

- After `A=1`, the cache block will propagate to T_1
  - Same for `B=1`
- So what values are possible for (r_0, r_1)?

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

Thread 0 “Executes” First

r_0	r_1	Execution Order
0	1	(S_0) (L_0) (S_1) (L_1)

Thread 0

```
// A = r₀ = 0

(S₀) A = 1;
(L₀) r₀ = B;

print(r₀);
```

Thread 1

```
// B = r₁ = 0

(S₁) B = 1;
(L₁) r₁ = A;

print(r₁);
```

Thread 1 “Executes” First

r_0	r_1	Execution Order
0	1	(S_0) (L_0) (S_1) (L_1)
1	0	(S_1) (L_1) (S_0) (L_0)

Thread 0

```
// A = r_0 = 0

(S_0) A = 1;
(L_0) r_0 = B;

print(r_0);
```

Thread 1

```
// B = r_1 = 0

(S_1) B = 1;
(L_1) r_1 = A;

print(r_1);
```

Stores “Execute” First

r_0	r_1	Execution Order
0	1	(S_0) (L_0) (S_1) (L_1)
1	0	(S_1) (L_1) (S_0) (L_0)
1	1	(S_1) (S_0) (L_1) (L_0)

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

Loads “Execute” First??

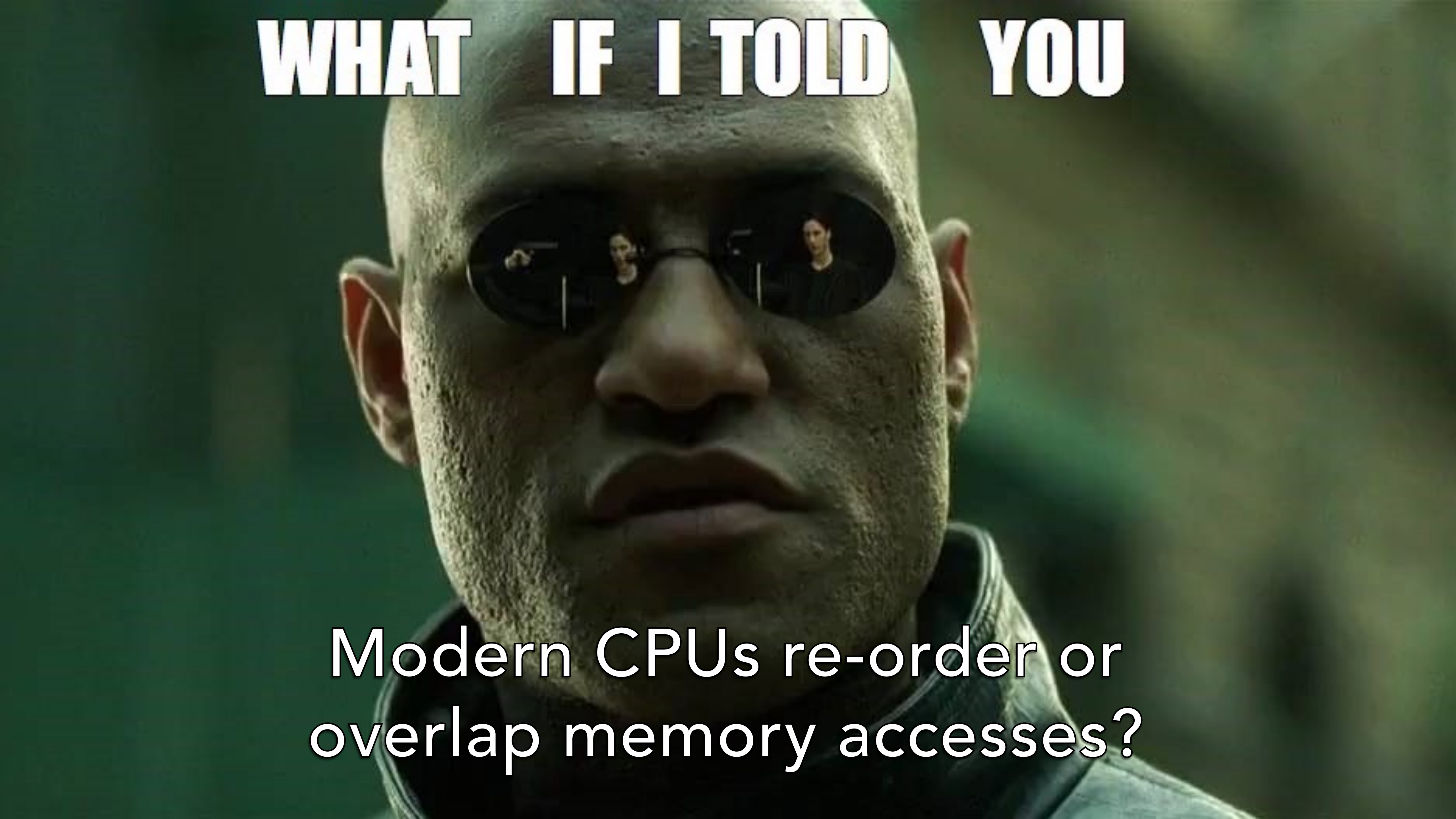
r_0	r_1	Execution Order
0	1	$(S_0)\ (L_0)\ (S_1)\ (L_1)$
1	0	$(S_1)\ (L_1)\ (S_0)\ (L_0)$
1	1	$(S_1)\ (S_0)\ (L_1)\ (L_0)$
0	0	$(L_1)\ (L_0)\ (S_1)\ (S_0)$

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```



Modern CPUs re-order or overlap memory accesses?

Loads “Execute” First??

- Yes, reading (0,0) is possible in the majority of today’s CPUs
  - And, furthermore, **this still satisfies cache coherence!**
- How?

Thread 0

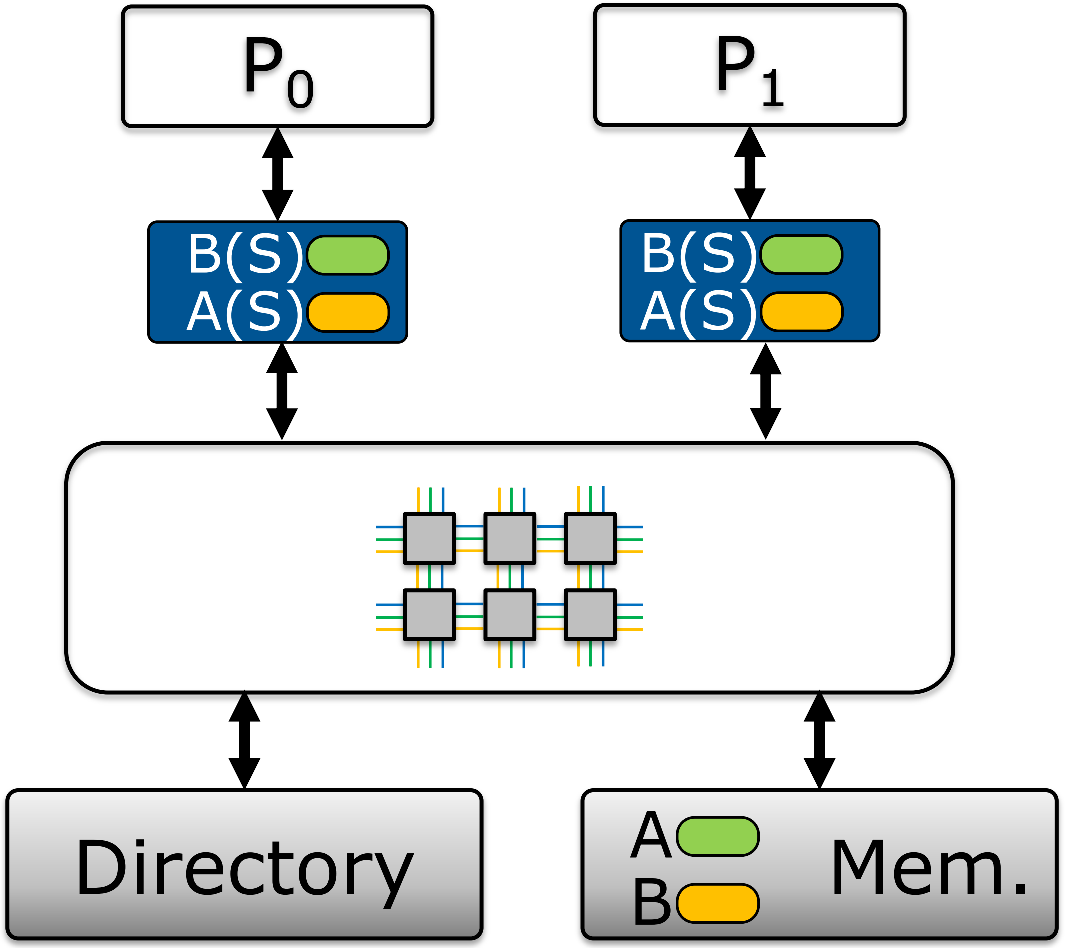
```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

A Sample Cache-Coherent Execution

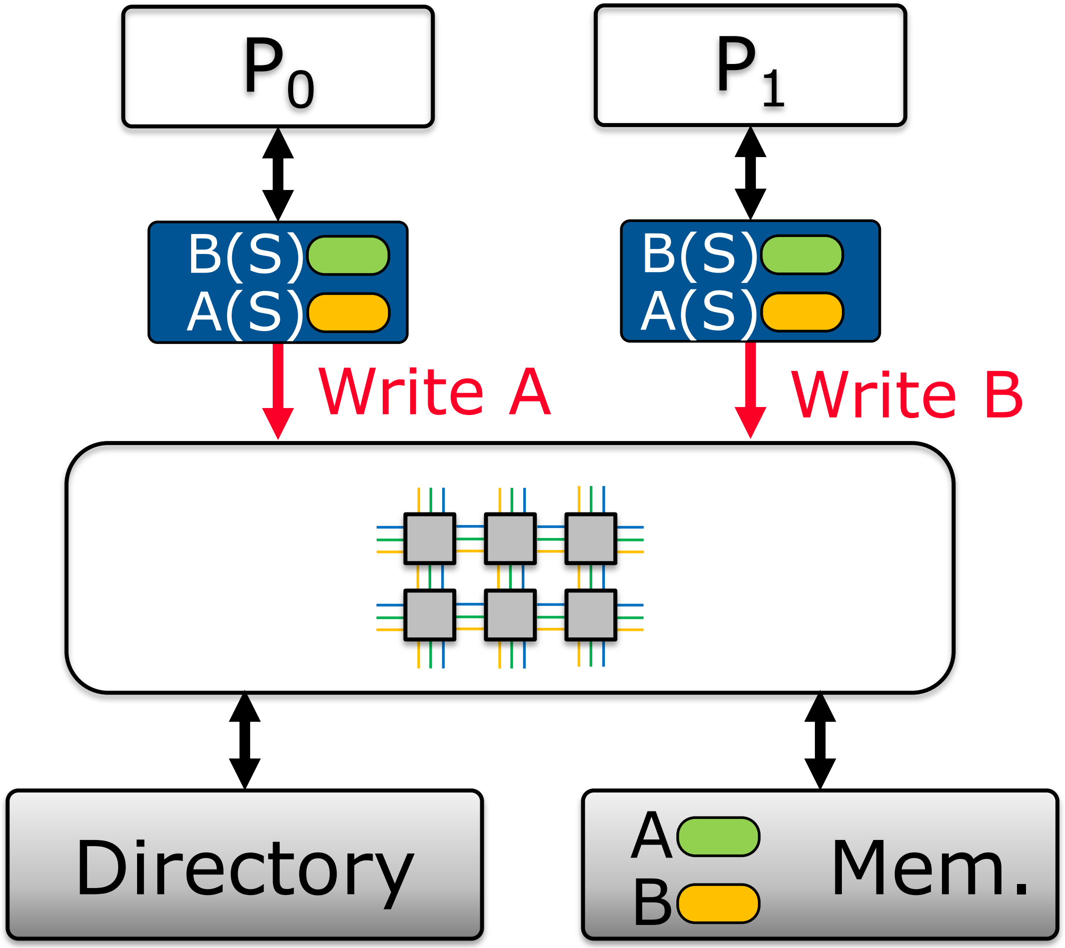
- Both processors begin `A & B` in shared state



A Sample Cache-Coherent Execution

- Both processors begin `A & B` in shared state

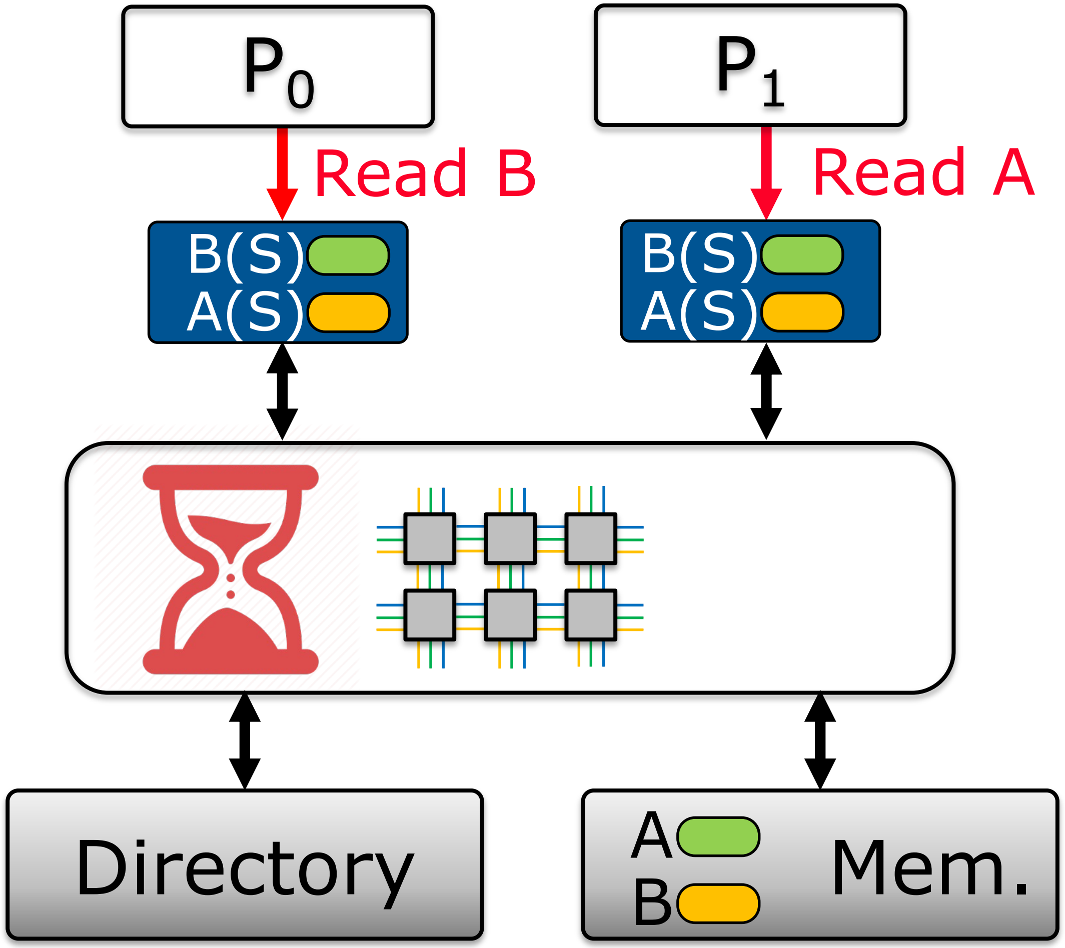
1. CPUs issue the stores simultaneously
   - Generate 2x Writes
   - Requests injected into network, headed for the directory



A Sample Cache-Coherent Execution

- Both processors begin `A & B` in shared state

1. CPUs issue the stores simultaneously
2. CPUs issue reads
   - Non-blocking caches proceed while msgs. are in network
   - Reads hit
   - Return: `A = B = 0`



Why Allow this Behavior?

- All system stakeholders have different needs!
- Programmers:
  - Want memory to behave like everything happens in program order, and atomically → Easy to reason about
  - But…. they want it to be fast!
- H/W Designers:
  - Want the ability to re-order operations for performance
  - e.g., the example we just saw

- Therefore, we must define a rigorous memory model to tell each layer what can and can’t happen

Intuitive Expectation for Shared Memory

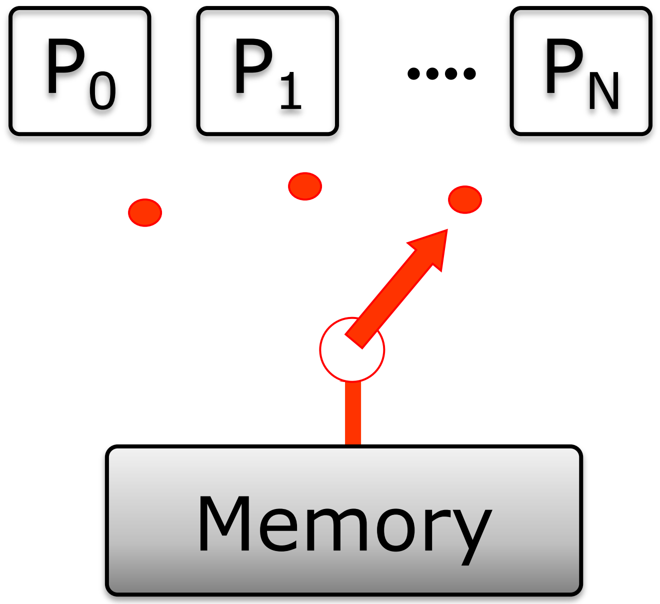
- Called Sequential Consistency (SC)
  - MP should behave like "multitasked" single core
  - (Earned Leslie Lamport the Turing Award in 2013!)

- More formally, a MP is SC if:
  - "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program."

Sequential Consistency

- Memory appears like it has a “switch” in front
  - Executes each processor’s memory accesses atomically and in program order

- Therefore:
  - Memory has one seq. order
  - It represents some interleaving of each processor’s operations
  - ✓ Therefore, we are SC



Exercise: Same Example With SC

- Assuming SC, what values can be printed now?
  - Hint, definition says:
    - Operations from all threads happen sequentially
    - The memory sees the operations in program order

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

Exercise: Same Example With SC

- Answer:

r_0	r_1	Execution Order
0	1	(S_0) (L_0) (S_1) (L_1)
1	0	(S_1) (L_1) (S_0) (L_0)
1	1	(S_1) (S_0) (L_1) (L_0)

Thread 0

```
// A = r0 = 0

(S0) A = 1;
(L0) r0 = B;

print(r0);
```

Thread 1

```
// B = r1 = 0

(S1) B = 1;
(L1) r1 = A;

print(r1);
```

Implementing SC

- We have to ensure two things for SC:
  1. Memory accesses happen in program order
  2. Memory operations appear atomic (i.e., instantaneous to other processors in the system)

- Single core program order
  - Ensure that Load(A) reads the last Stored value
  - We will focus on this problem first

Reminder: Basic Out-of-Order CPU

- Enables instructions to execute out of order
  - Why? Expose instruction-level-parallelism (ILP)
- In following example, assume I_1, I_3 miss in LLC:
  - Loads take 100 cycles, adds take 1 cycle
  - In an in-order core, have to wait for the values in r_2,r_4

Inst.	Code	In-Order Cycle
1	load r2, [r3]	100
2	add r2, r2, 4	101
3	load r4, [r5]	201
4	add r4, r4, 4	202
5	add r6, r2, r4	203

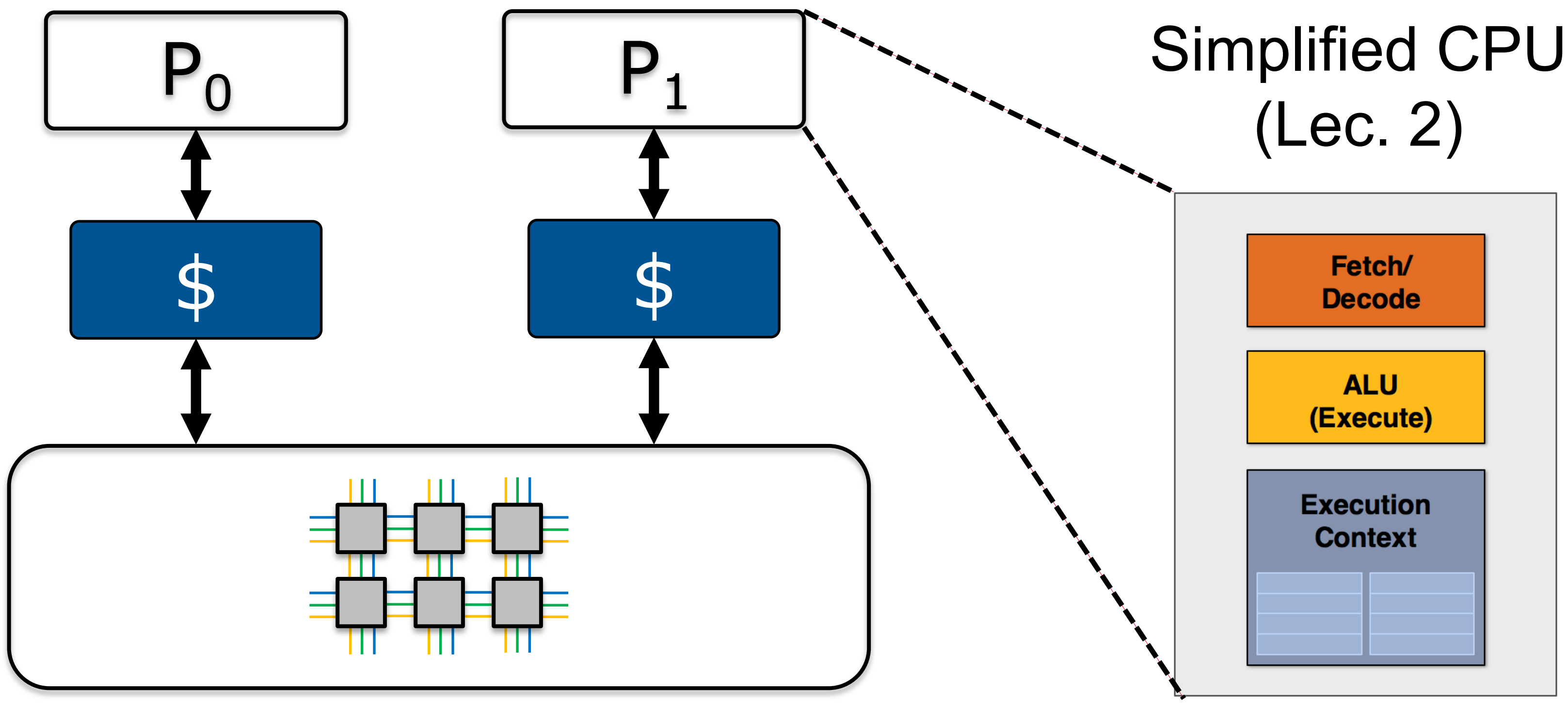
Reminder: Basic Out-of-Order CPU

- Enables instructions to execute out of order
  - Why? Expose instruction-level-parallelism (ILP)
- In following example, assume $I_{1,}I_3$ miss in LLC:
  - Loads take 100 cycles, adds take 1 cycle
  - OoO core overlaps the loads, as they are independent

Inst.	Code	In-Order Cycle	OoO Cycle
1	load r2, [r3]	100	100
2	add r2, r2, 4	101	101
3	load r4, [r5]	**201**	**102**
4	add r4, r4, 4	202	103
5	add r6, r2, r4	203	104

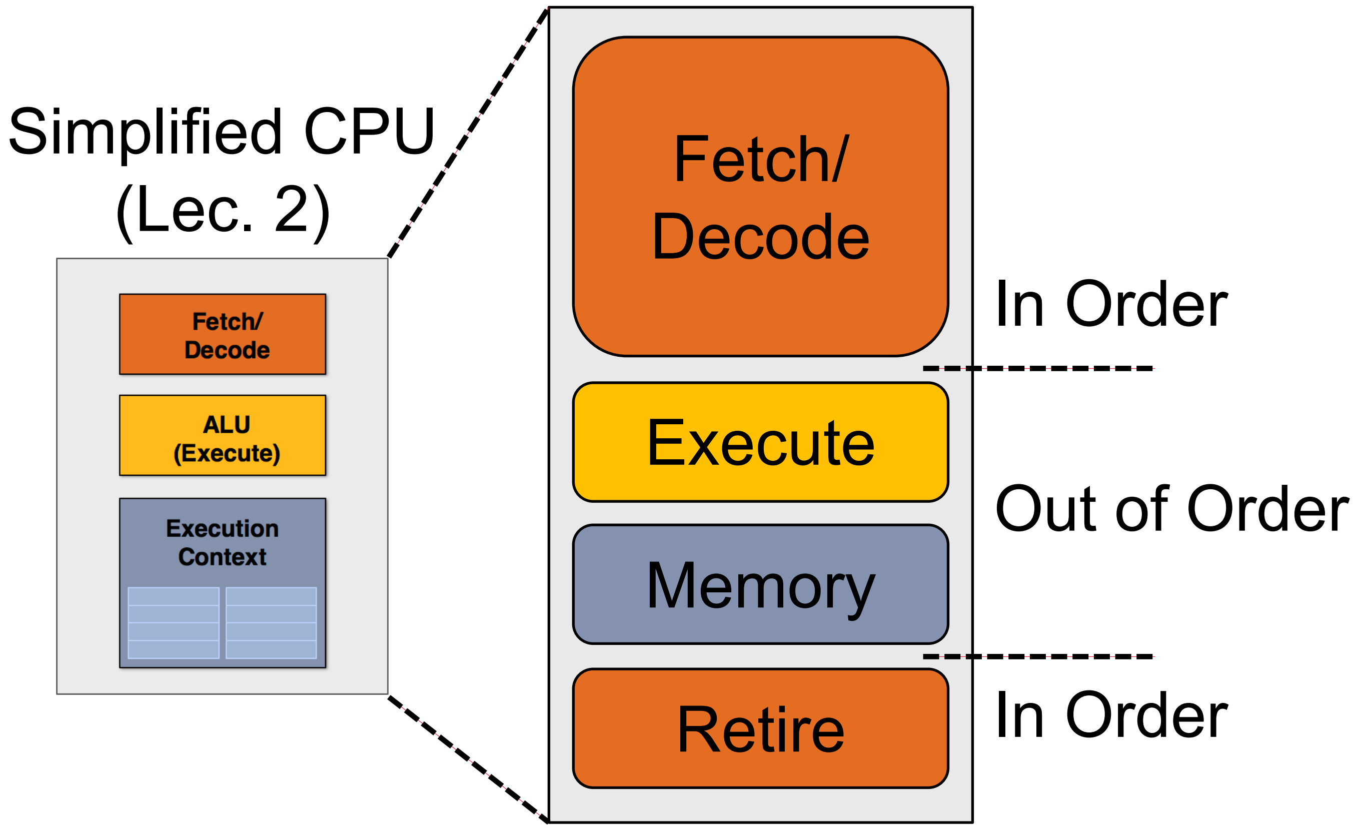
Reminder: Inside CPU

◆ Fetch/Decode, Execute, Keep context



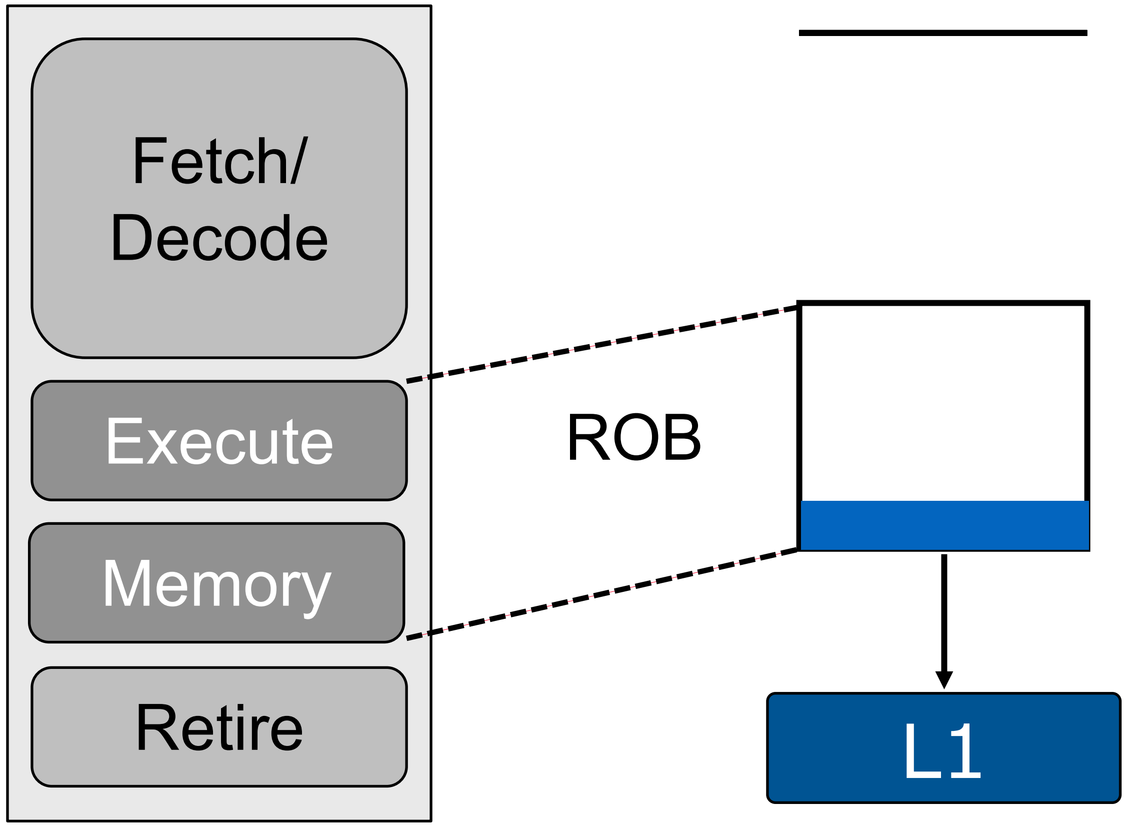
Reminder: OoO CPU Pipeline

- Fetch instructions in order, execute out of order, but reconstruct order when retiring
  - Why? Preserve CPU state on trap or exception



Reminder: OoO CPU Pipeline

- Central structure called the "Reorder Buffer"
  - Instructions enter in order and exit (retire) in order



Reminder: Register Dependences

- Register names are encoded in the instruction
- Register dependences are established at **decode**
- All dependences among instructions are established in program order

Inst.	Code	Dependence
1	load r2, [r3]	produces r2
2	add r2, r2, 4	gets r2 from I_1, produces r2
3	load r4, [r5]	produces r4
4	add r4, r4, 4	gets r4 from I_3, produces r4
5	add r6, r2, r4	gets r2 from I_2, gets r4 from I_4, produces r6

But What About Memory Dependence?

- **Note:** This is non-trivial with the memory ops!
  - Inst. 1 is a load miss (will take 100 cycles)
  - Don’t know addresses of 4(r1) and 8(r5) until execute
  - Addresses are not known at decode time

Inst.	Original
1	load r1, 0(r4)
2	store r2, 4(r1)
3	load r3, 8(r5)

Is 4(r1) the same address as 8(r5)?

But What About Memory Dependence?

case #1 4(r1) = 8(r5)

Inst.	Original
1	load r1, 0(r4)
2	store r2, 4(r1) waiting for r1
3	load r3, 8(r5)

- `load` has to wait for `store` to complete
- `r3` will get the same value stored in `r2`

case #2 4(r1) != 8(r5)

Inst.	Original
1	load r1, 0(r4)
2	store r2, 4(r1) waiting for r1
3	load r3, 8(r5)

- Inst. 3 is independent of Inst. 2
- Inst. 2 could go ahead but does not know until Inst. 1 finishes and `4(r1)` becomes known at execute

Need to Order Memory Instructions

- Need to do the following things:
  - Track the FIFO program order of loads & stores
  - Resolve addresses when they are ready
  - On a load, check for the **youngest** store to this address
- Use a structure called a “Load-Store Queue” (LSQ)

Multiprocessor Memory Consistency

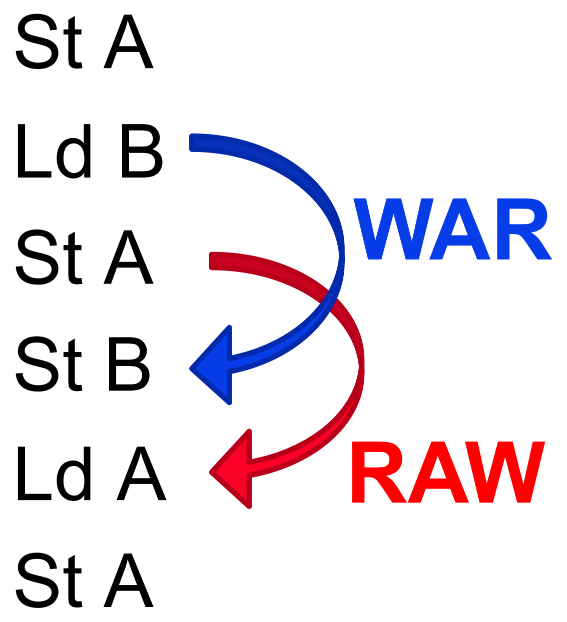
- Need a uniprocessor memory order
  - Make sure that a load and store in program order pass their values correctly (even in sequential, single-threaded programs)
- Need multiprocessor memory order
  - Programmers want all memory accesses to be atomic & in-program order
- **Memory consistency model affects performance**
  - Dictates acceptable memory re-orderings

Types of Memory Dependences

$$\begin{tikzcd}
\text{St A} \\
\text{Ld B} \\
\text{St A} \arrow[dd, bend left=90, "\text{RAW}"] \\
\text{St B} \\
\text{Ld A} \\
\text{St A}
\end{tikzcd}$$

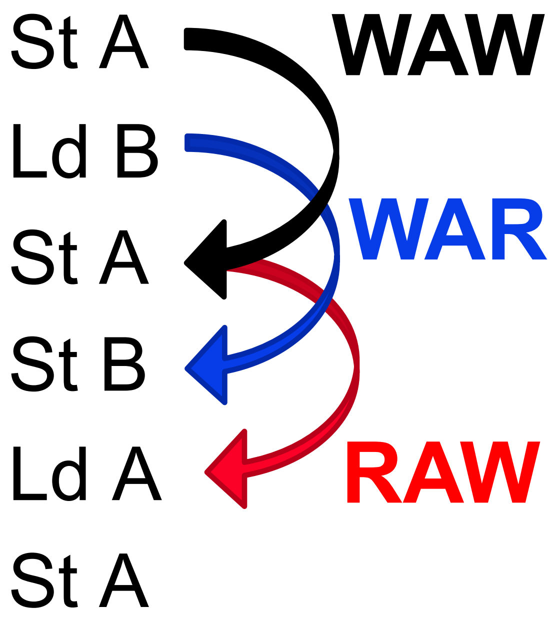
- Read After Write (RAW)
  - Store must complete before Load
  - Notation: W → R (write blocks read)

Types of Memory Dependences



- Read After Write (RAW)
  - Store must complete before Load
  - Notation: W → R (write blocks read)
- Write After Read (WAR)
  - Load must complete before Store
  - Notation: R → W (read blocks write)

Types of Memory Dependences



- Read After Write (RAW)
  - Store must complete before Load
  - Notation: W → R (write blocks read)
- Write After Read (WAR)
  - Load must complete before Store
  - Notation: R → W (read blocks write)
- Write After Write (WAW)
  - Store must complete before Store
  - Notation: W → W (write blocks write)

Specification for Solving Memory Dependences

(“<<“ means precedes)

- Given $\text{Store}_i(A, V) \ll \text{Load}_j(A)$
  - $\text{Load}_j(A)$ must return V if there isn’t a $\text{Store}_k()$ where:
    $\text{Store}_i(A, V) \ll \text{Store}_k(A, V') \ll \text{Load}_j(A)$

- Can guarantee by observing these dependences:
  - RAW: $\text{Store}(A,V) \rightarrow \text{Load}(A)$
  - WAW: $\text{Store}(A,V) \rightarrow \text{Store}(A,V')$
  - WAR: $\text{Load}(A) \rightarrow \text{Store}(A,V')$

Take a break!

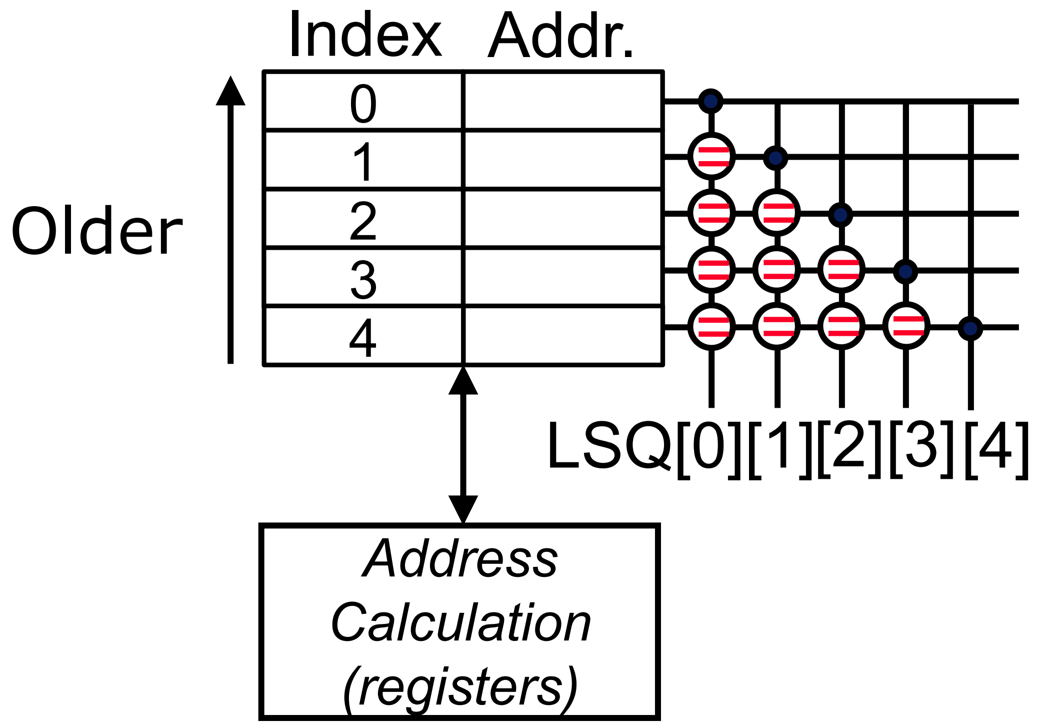
Load Store Queue Functionality

- LSQ accomplishes the following two key tasks:
  1. Resolve which Ld/St addresses overlap
  2. Hold all store operations until they retire

- Address resolution necessary to forward values
- Cannot write “speculative” values to caches
  - Speculative values are those that ran out of order
  - They wait until all prior accesses are complete
  - Otherwise, they may corrupt the system’s state

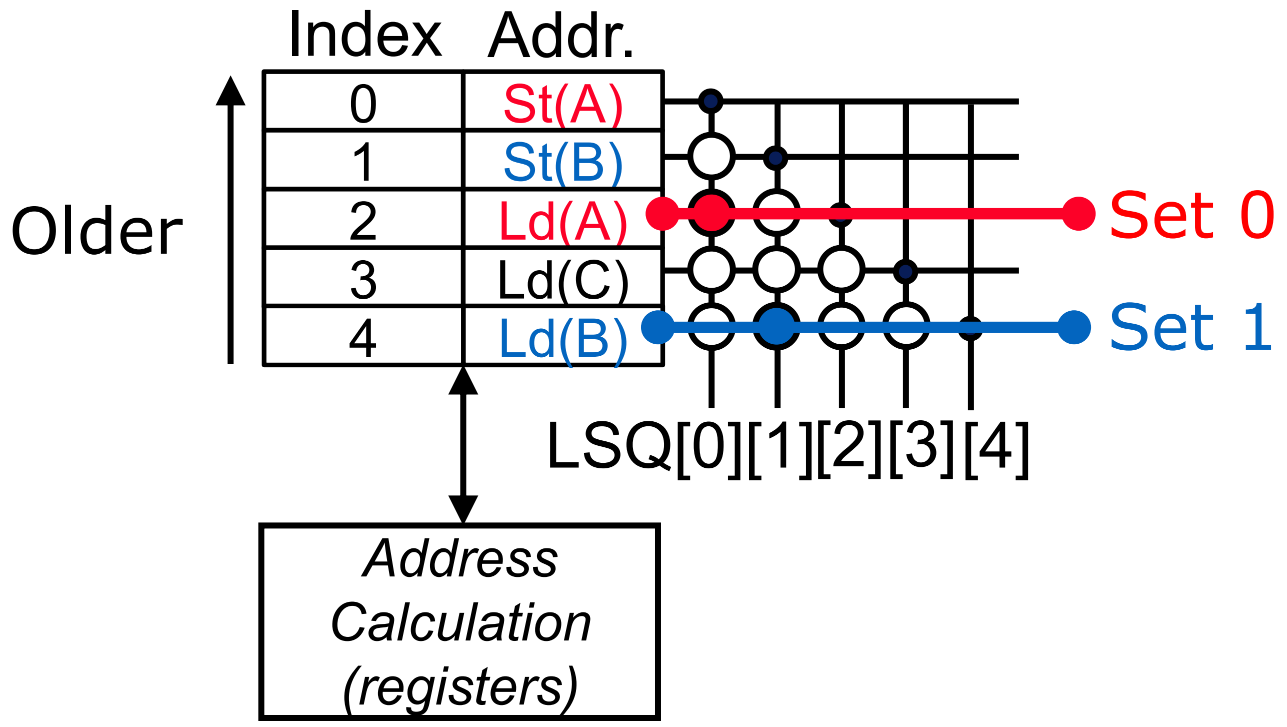
Load Store Queue Address Resolution

- Address Resolution in LSQ
  - Use an NxN half-matrix of comparators, cross checks every entry against all **older** ones



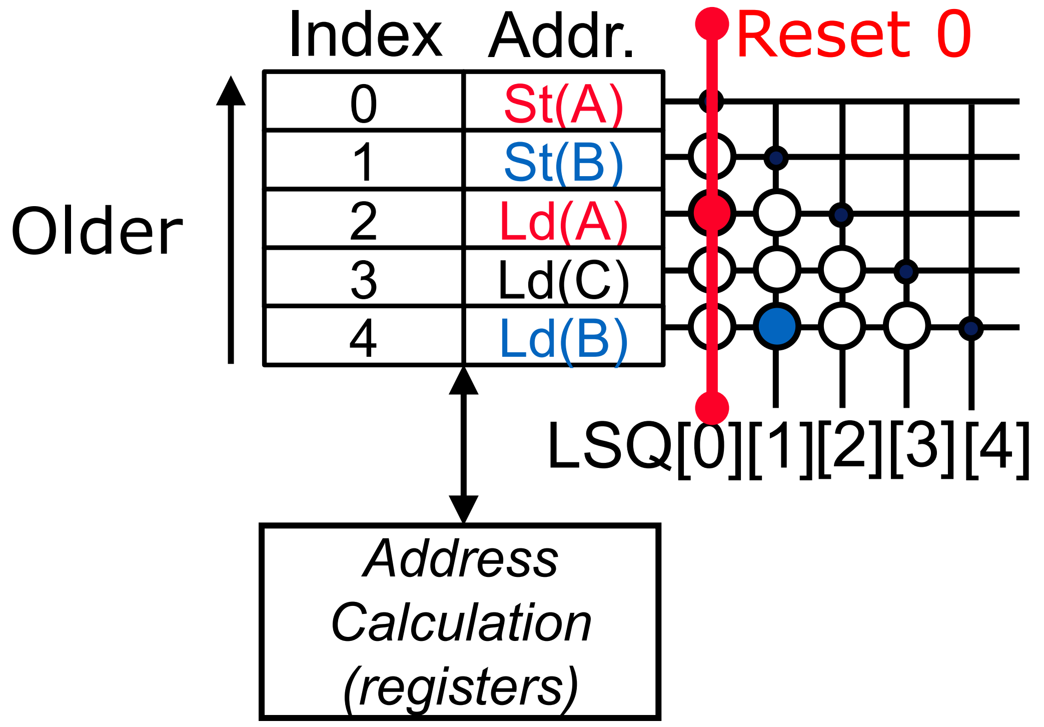
Load Store Queue Address Resolution

- Address Resolution in LSQ
  - Load sets bits for **all** older stores they depend on



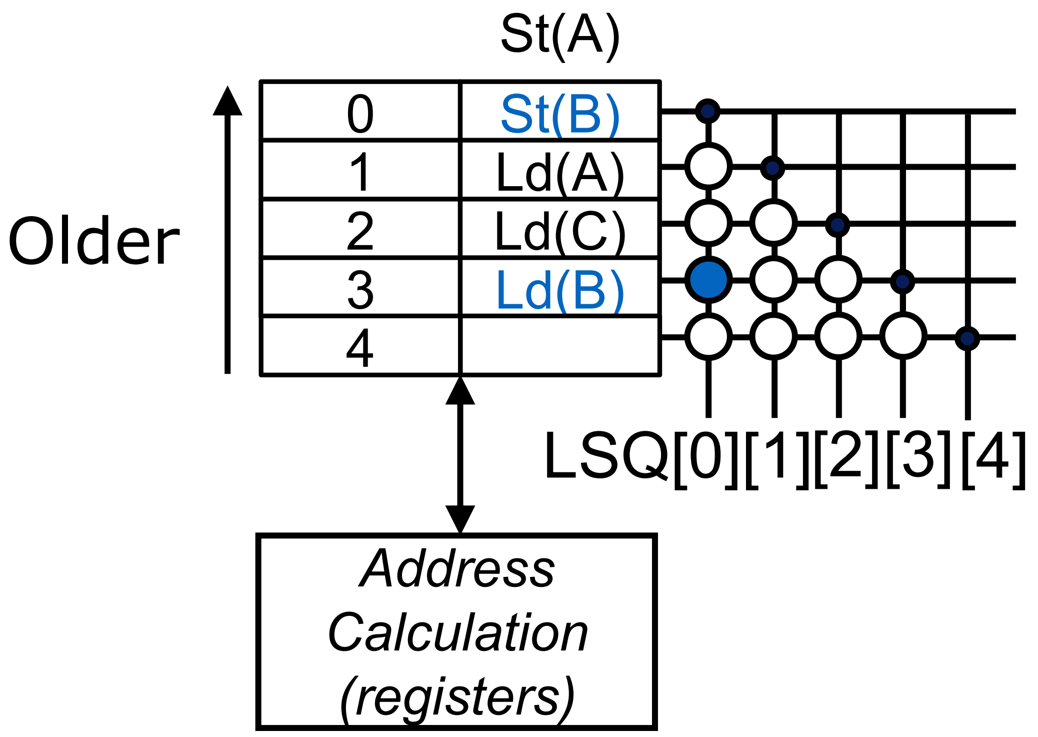
Load Store Queue Address Resolution

- Address Resolution in LSQ
  - Load sets bits for **all** older stores they depend on
  - Store resets its column when updating the cache



Load Store Queue Address Resolution

- Address Resolution in LSQ
  - Load sets bits for **all** older stores they depend on
  - Store resets its column when updating the cache



Load Store Queue Functionality

- Hold speculative stores until they resolve
  - OoO processors predict branches and speculate
  - e.g., Store r3, 0(r4) but preceding branch was mispredicted!
- May corrupt memory if store allowed to complete

```
LOOP:
    …
    test r2,0         # test loop counter
    jmp_nz LOOP       # loop if not zero yet
    store r3,0(r4)    # offending store
```

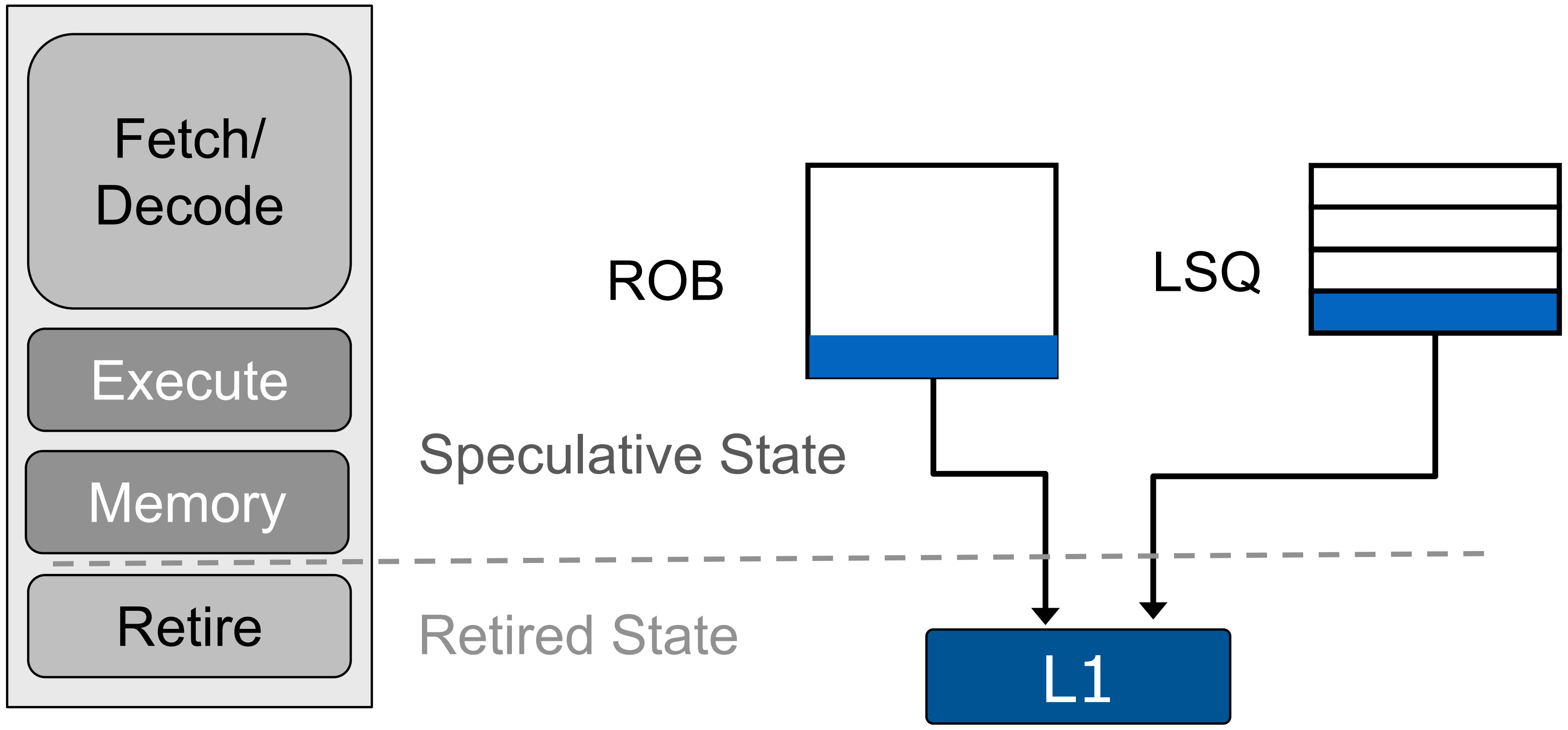
How to Block Speculative Stores

- Integrate LSQ operation w. Reorder Buffer (ROB)
  - Reminder: Instructions get an ROB entry at rename, and release it when they commit
  - In-order fetch, In-order commit, OoO execute

- Only remove an LSQ entry when store exits ROB

How to Block Speculative Stores

- Memory Operation (blue) at head of ROB
  - Retires from ROB and de-allocates LSQ entry
  - Has not triggered exception or page walk

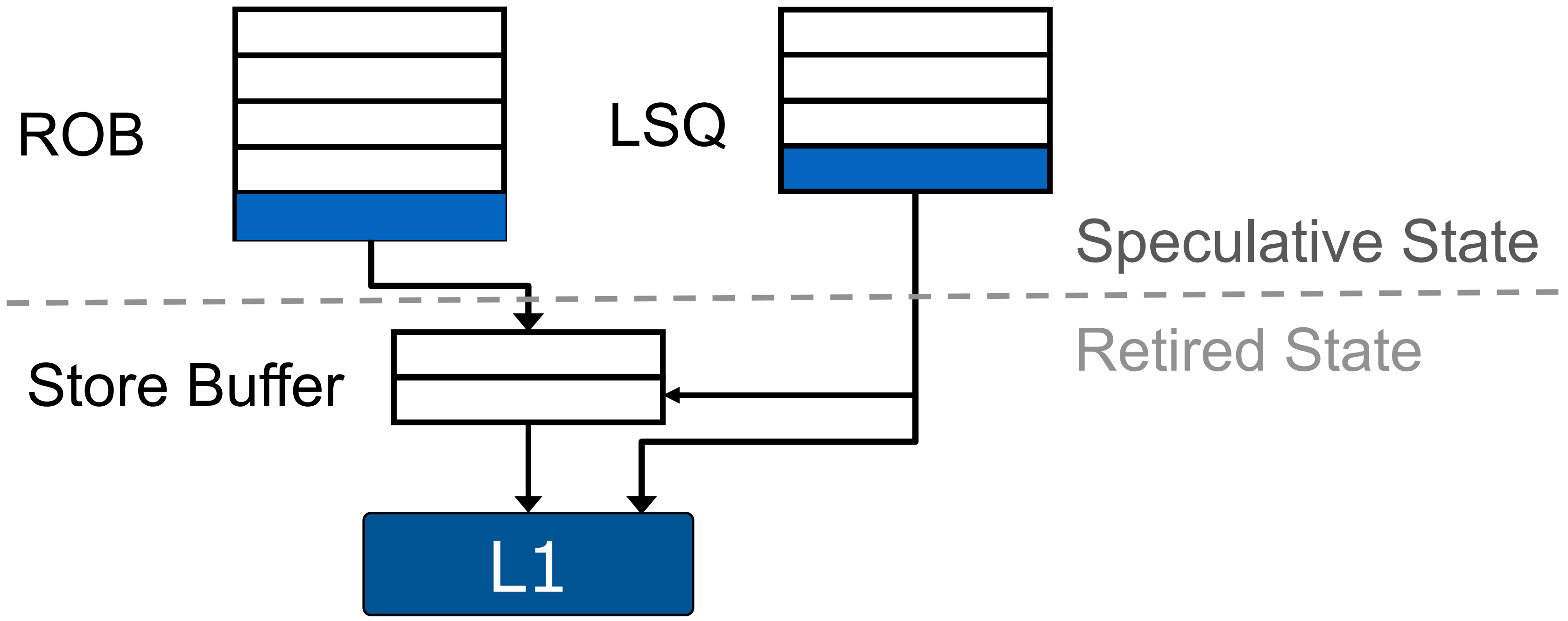


Problem: Why wait for stores?

- Currently, processor waits for stores to complete
  - What if it misses in L1/L2/LLC? **100+ cycle stall**
  - LSQ entries are scarce due to NxN address dep. check
- Stores do not generate operands for the core
  - Loads and arithmetic operations do
- Processor should continue while store pending!
  - Reclaim LSQ entry for new memory operations

Solution: Add Store Buffer

- Store buffer (SB) sits between core and L1 cache
  - Holds committed stores, which cannot be rolled back
- **Note:** Now loads must check SB as well as L1
  - No guarantee on when values will be written



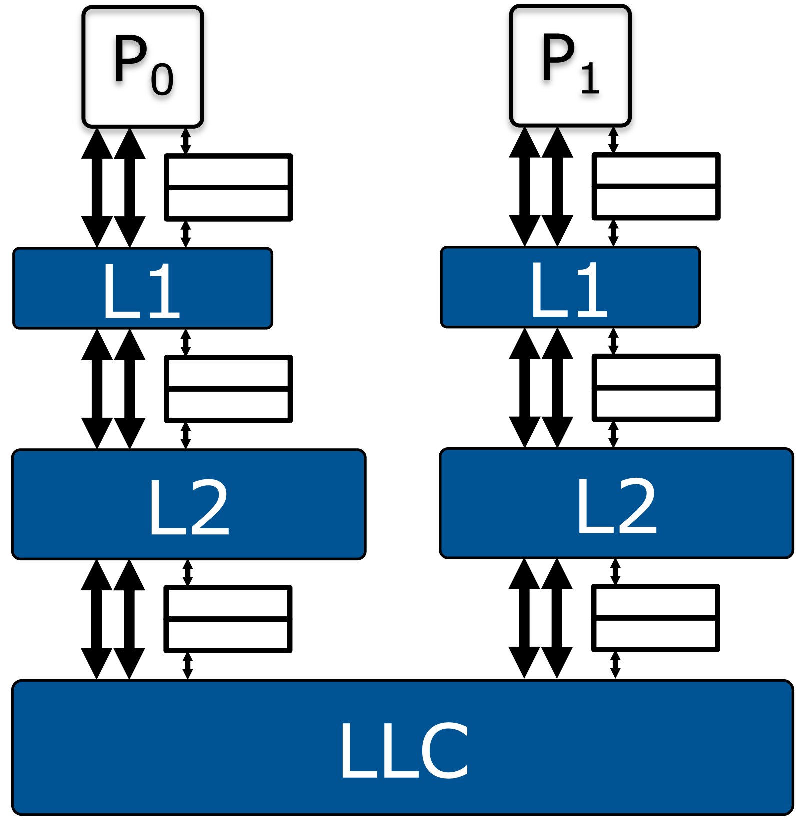
Return to Conditions for SC

- Maintain program order in the CPU
  - Use LSQ/SB, in the fashion we just demonstrated

- **Must** maintain atomicity in the memory system
  - In small-scale systems, use a shared bus
  - At larger scale, explicit completion acknowledgements

Problems with the SC Model

- True SC would be painfully slow
  - Can only issue **one** memory operation at a time
- Modern processors are:
  - Superscalar
  - Out of order
- Caches are:
  - Non-blocking, multi-ported
  - Buffered at input/output

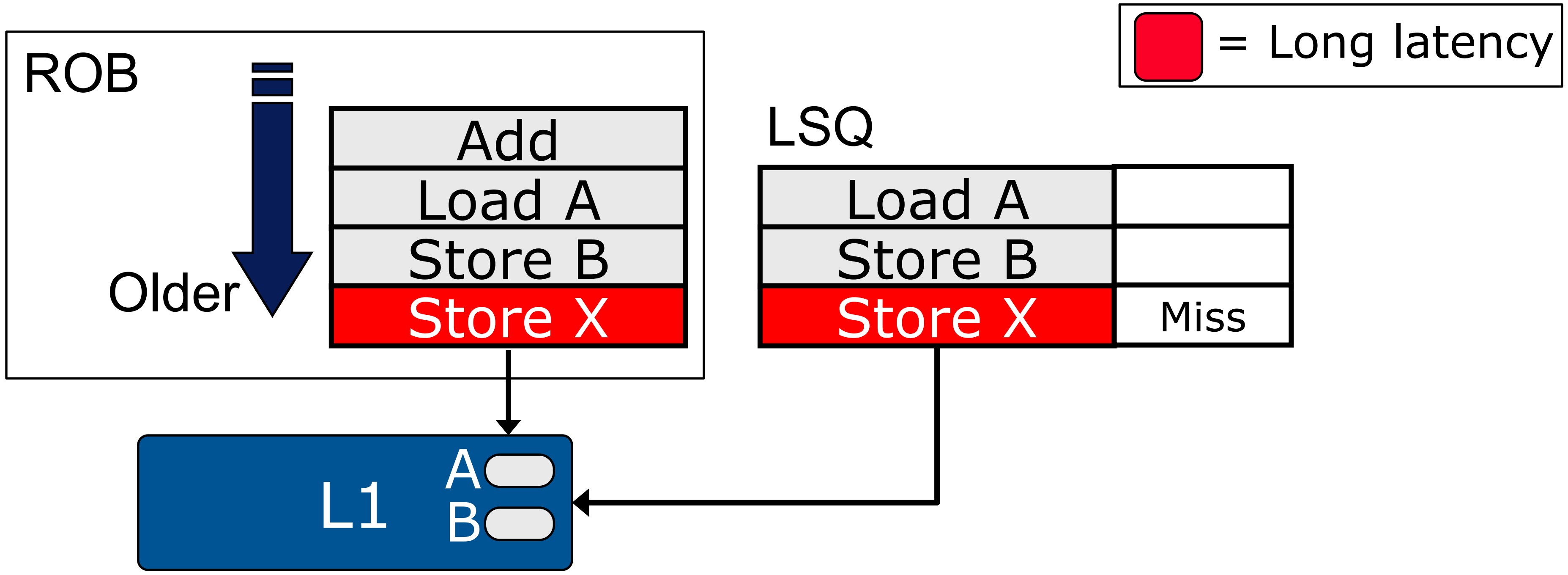


Example

- To quantify SC performance, use this code:

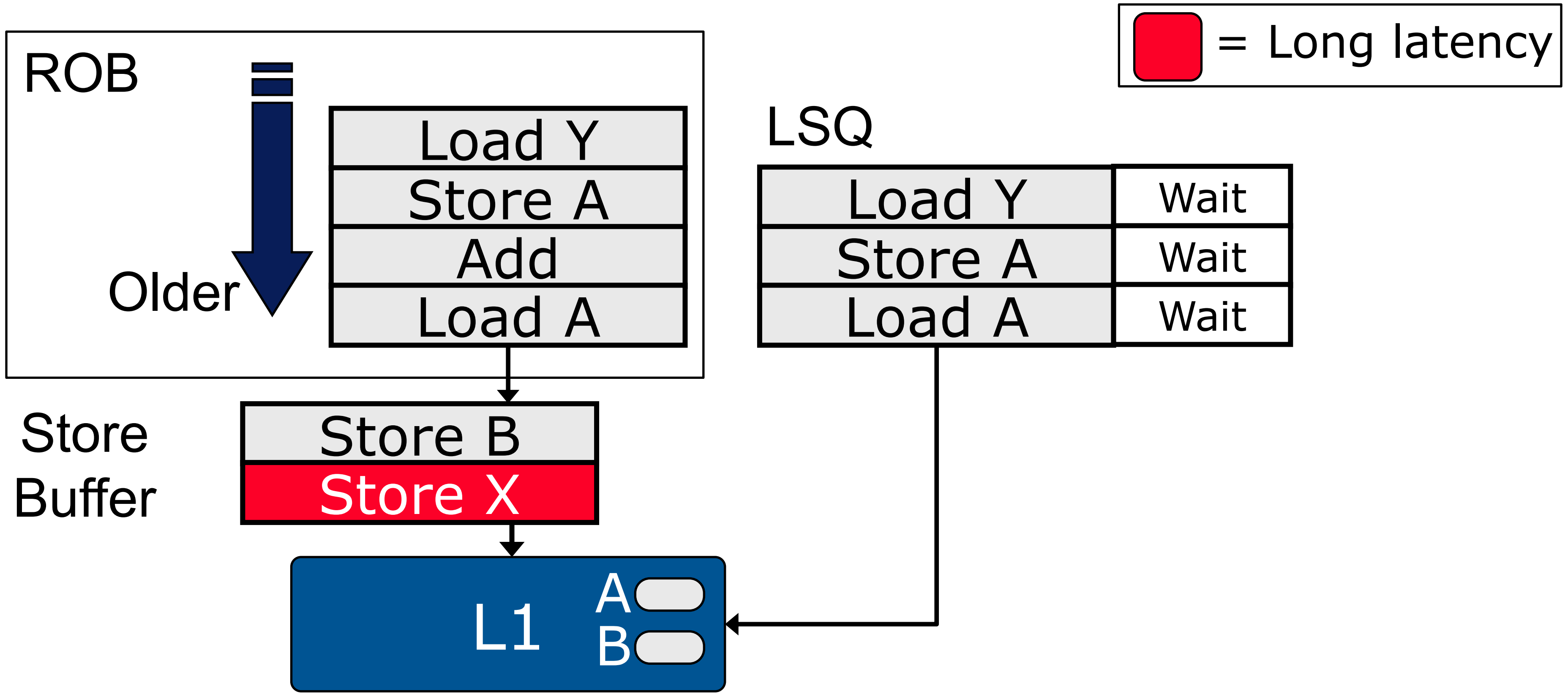
```
Store X         ; misses in L1 & L2
Store B         ; hits in L1
Load A          ; hits in L1
Add
Store A
Load Y          ; misses in L1 & L2
Store Z         ; misses in L1 & L2
```

Execution with Naïve SC



- Store X blocks ROB
  - Cannot overlap Load Y & Store Z (other misses)

Execution with SC + Store Buffer



- Store buffer frees up two slots in ROB/LSQ
  - All ops still need to wait on Store X

Performance Comparison

- Assume all misses take 100 cycles, hits 1 cycle

	behavior	**Naïve SC**
Store X	L2 miss	100
Store B	L1 hit	1
Load A	L1 hit	1
Add	-	1
Store A	L1 hit	1
Load Y	L2 miss	100
Store Z	L2 miss	100
Total	**-**	**304**

Using Stalled Operations in LSQ

- If operation in LSQ has address, why not issue it?
  - Violates order or atomicity
- Idea: Peek at L1, see if address is already there
  - If not (miss), fetch the block from lower level into L1
  - If so, do **not** load the value into the core
- Insight:
  - Fetching blocks into L1 from lower levels (or other L1's) does not impact order or atomicity
  - No values move between the core & L1
- Helps overlap latency
  - Can fetch as many blocks in parallel into L1 as needed

New SC Interpretation

- Memory appears program order and atomic
  - All loads/stores still execute with no re-ordering
- But, we add the ability to peek into L1 cache
  - No ordering or atomicity constraints
  - Other cores may see coherence messages!
  - e.g., Load(A) invalidates A in remote core

Exercise: Does Peeking Violate SC?

- Same example as before, with SC + L1 peeking
  - Assume 2 cores, private L1 caches, w. bus interconnect
- Can we possibly observe $r_1 = r_2 = 0$?

Thread 0

```
// A = r₁ = 0

(S₀) A = 1;
(L₀) r₁ = B;

print(r₁);
```

Thread 1

```
// B = r₂ = 0

(S₁) B = 1;
(L₁) r₂ = A;

print(r₂);
```

Exercise: Does Peeking Violate SC?

- Same example as before, with SC + L1 peeking
  - Assume 2 cores, private L1 caches, w. bus interconnect
- Can we possibly observe $r_1 = r_2 = 0$?
- Answer: No.
  - If Load(B) peeks, value brought into cache is B = 1 or B = 0 (depending on T1). Still needs to read B when it executes (S1 may have invalidated it)

Thread 0

```
// A = r1 = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

Thread 1

```
// B = r2 = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

Exercise: Does Peeking Violate SC?

Highly Recommended

Do the same exercise for a general non-atomic interconnect w. directory protocol, convince yourself that we still cannot see $r_1 = r_2 = 0$!

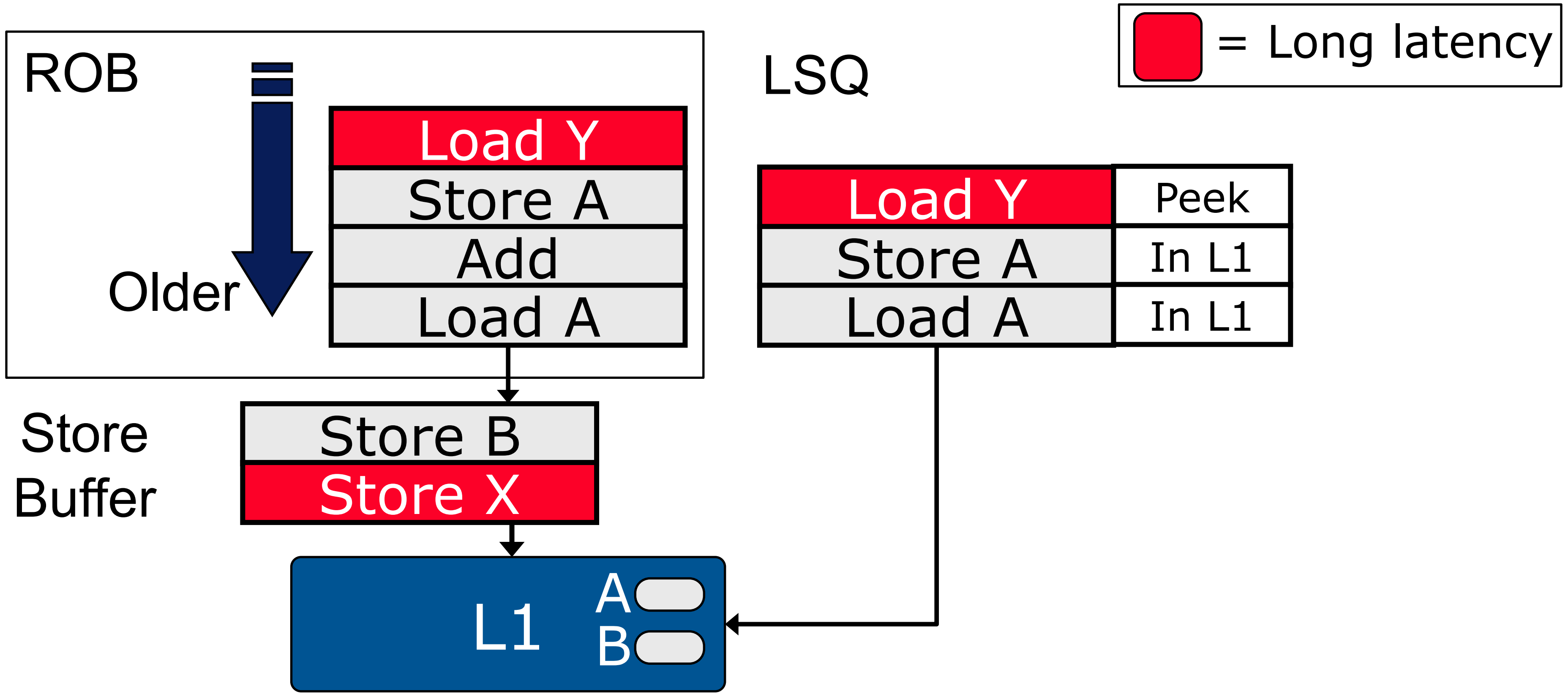
Thread 0

```
// A = r₁ = 0

(S₀) A = 1;
(L₀) r₁ = B;

print(r₁);
```

Thread 1

```
// B = r₂ = 0

(S₁) B = 1;
(L₁) r₂ = A;

print(r₂);
```

Execution with SC + SB + L1 Peeking



- While waiting, peek on all waiting ops. in LSQ
  - A is in cache, Y is not → Overlap latency of Load Y

Performance Comparison Continued

- Loading Y in advance converts miss latency to hit

	behavior	Naïve SC	SC + Peek
Store X	L2 miss	100	100
Store B	L1 hit	1	1
Load A	L1 hit	1	1
Add	-	1	1
Store A	L1 hit	1	1
Load Y	L2 miss	100	1
Store Z	L2 miss	100	97
Total		**304**	**202**

Unblocking the LSQ and ROB

- Peeking in L1 does not let ops. proceed in CPU
  - ROB is completely full, no instructions can fetch
  - Why? Load A waits for Store X, Store B

- To keep the CPU running, need to free up Load A
  - Unfortunately, cannot do with SC
  - Violates program order constraint

- Idea: Relax W $\rightarrow$ R (write blocks younger read)
  - If address is the same, result comes from LSQ or SB
  - If different, let it pass and unblock the processor

Relaxed Consistency Models

- Defining a relaxed model requires the following:
  1. What specific orders are relaxed?
  2. Detectable by the programmer or compiler?
  3. Are there methods provided to disallow the behavior?

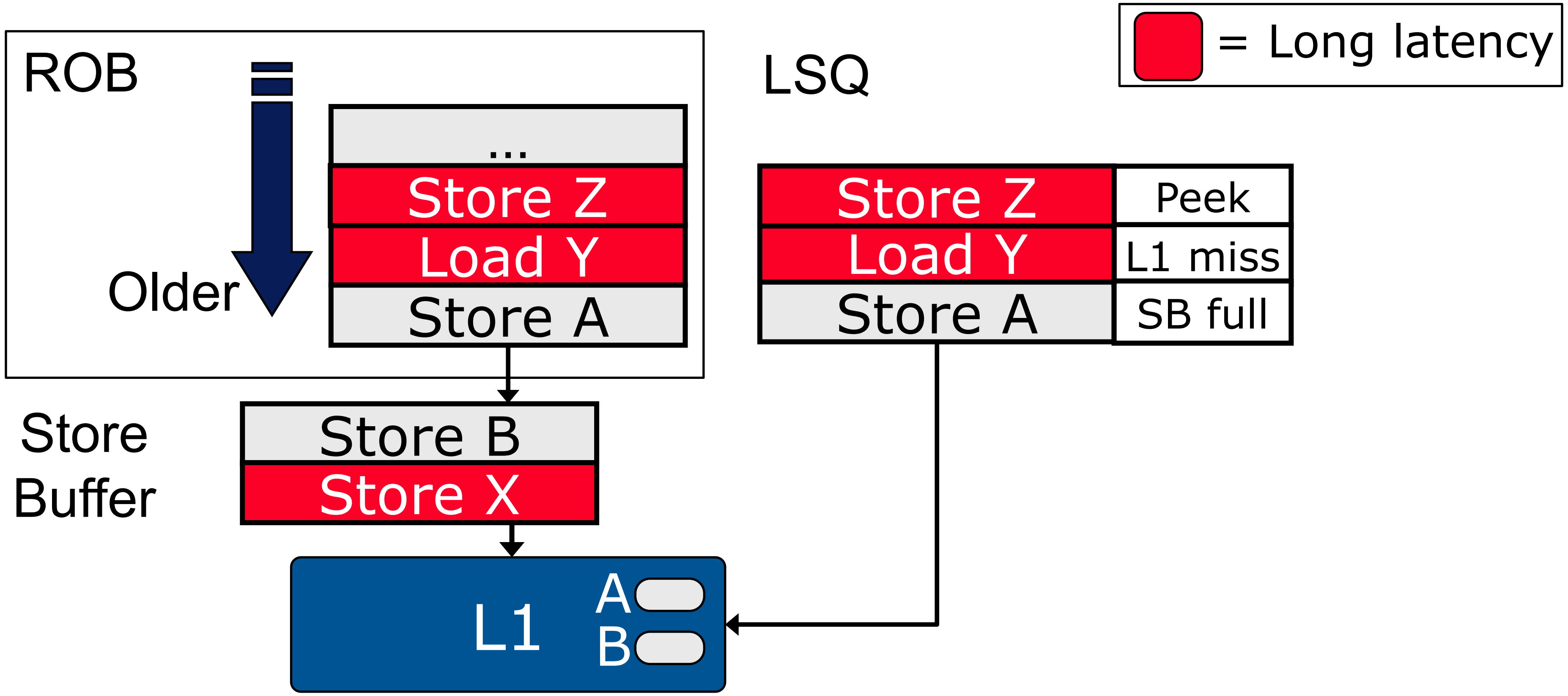
- In our idea to unblock independent reads:
  1. We relax W → R serialization
  2. Yes, it is detectable (try first example on your laptop!)
  3. Yes, but semantics depend…
     - e.g., MFence instructions (to come later)

Formally: Processor Consistency (PC)

- In VAX processors (‘70s) before being defined
- Specification:
  - ”Before a load is performed with respect to other processors, all preceding loads must be performed.”
  - “Before a store is performed with respect to other processors, all preceding operations (L & S) must be performed.”

- In plain language: Reads can bypass writes
- x86 (Intel/AMD) uses a variant of PC

Execution with PC



- Let Load A and Add retire from CPU
- Fetch, rename, and execute Store Z

Differences Between SC+Peek and PC

SC+Peek

Load Y	Peek
Store A	In L1
Load A	In L1

PC

Store Z	Peek
Load Y	L1 miss
Store A	SB full

Compare the two LSQs:

- In SC+Peek, Load A hits L1, but still had to wait
  - PC allows it to bypass the ordered Stores to X, B, and A
- In PC, why is Load Y an L1 miss, not a Peek?
  - Independent addresses, so the load is actually issued
  - If it hit the L1, another instruction could read the result

Performance Comparison Continued

- Assume all misses take 100 cycles, hits 1 cycle

	behavior	Naïve SC	SC + Peek	PC
Store X	L2 miss	100	100	100
Store B	L1 hit	1	1	1
Load A	L1 hit	1	1	1 (overlap)
Add	-	1	1	1 (overlap)
Store A	L1 hit	1	1	1
Load Y	L2 miss	100	1	1
Store Z	L2 miss	100	97	1
Total		**304**	**202**	**106**

PC Summary

- Ordering constraints relaxed: W → R
- PC provides a relaxed model whose semantics are relatively easy to reason about
  - Variants in most real CPUs (AMD, Intel, Oracle/Sun)

- How to enforce order in a relaxed memory model?
  - All ISAs have special **atomic** instructions (x86 – `xchg`)
  - We will study these in detail later in the course
  - Functionally, they transit the CPU, drain the entire store buffer, and the whole system can see them immediately

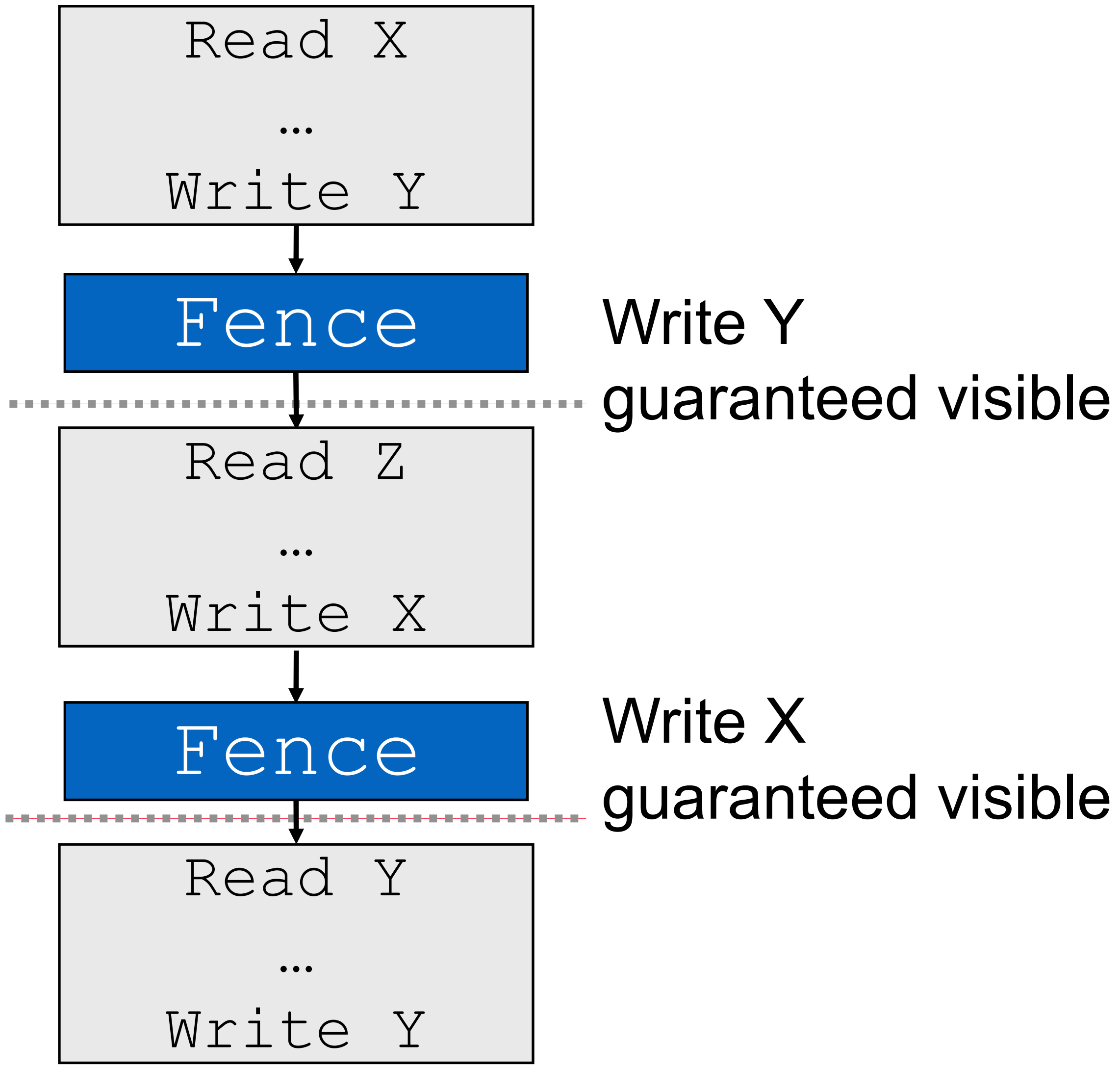
Further Ordering Relaxation

- Recall that PC blocks if SB is full or on an atomic
  - Given that SB size is limited, can only take PC so far
- Key constraint preserved in PC: R → RW
  - Reads block other reads and writes
- New idea: Relax everything, only obey uniprocessor constraints for correctness

Weak Consistency

- Memory ops. classified as data or synchronization
  - Only synchronization operations have any ordering
  - Data ops. have no order enforced among themselves
- Synch. instructions are called `Fences`
  - Enforces program order for operations before/after
- Weak Consistency is used in ARM, RISC-V

Conceptual Model for Weak Consistency

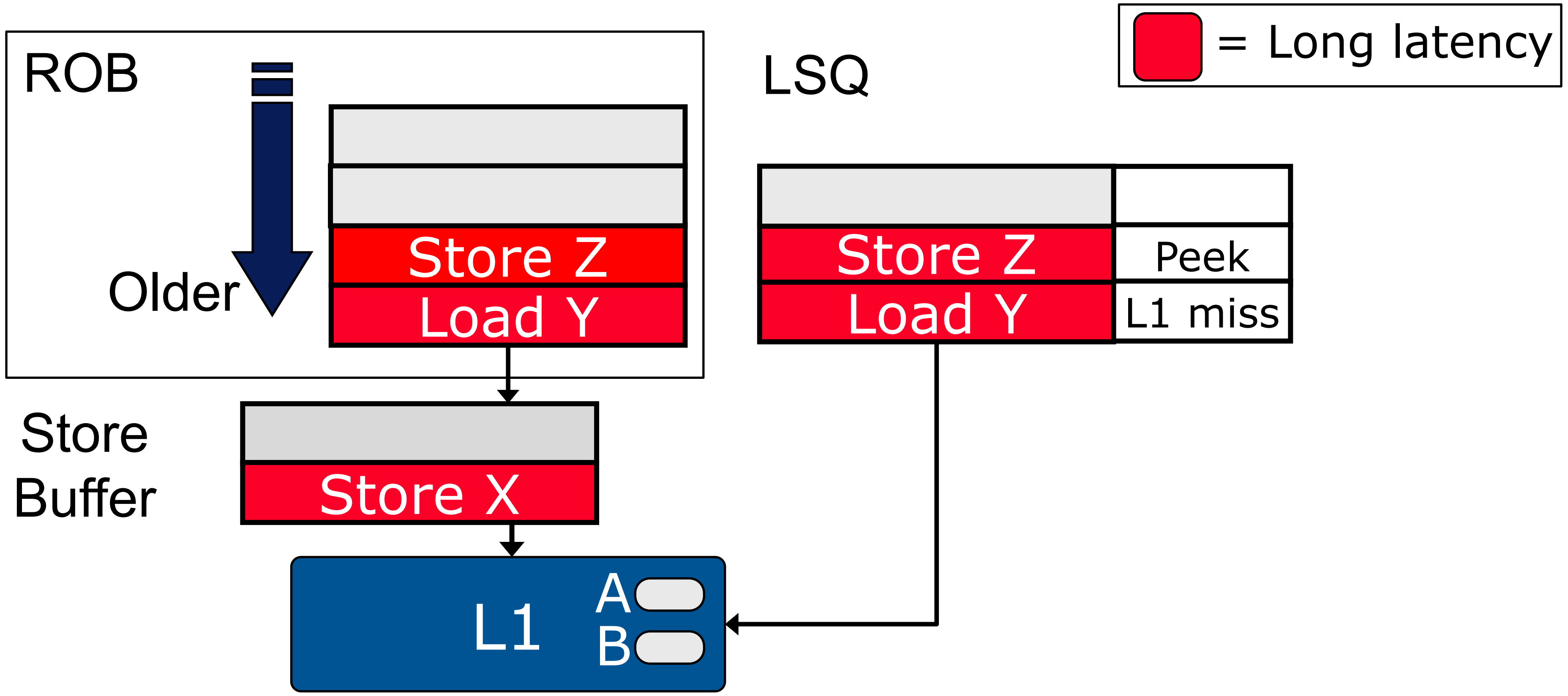


Example w. Weak Consistency

- Note: there are **no fences!**

```
Store X        ; misses in L1 & L2
Store B        ; hits in L1
Load A         ; hits in L1
Add
Store A
Load Y         ; misses in L1 & L2
Store Z        ; misses in L1 & L2
```

Execution with WC

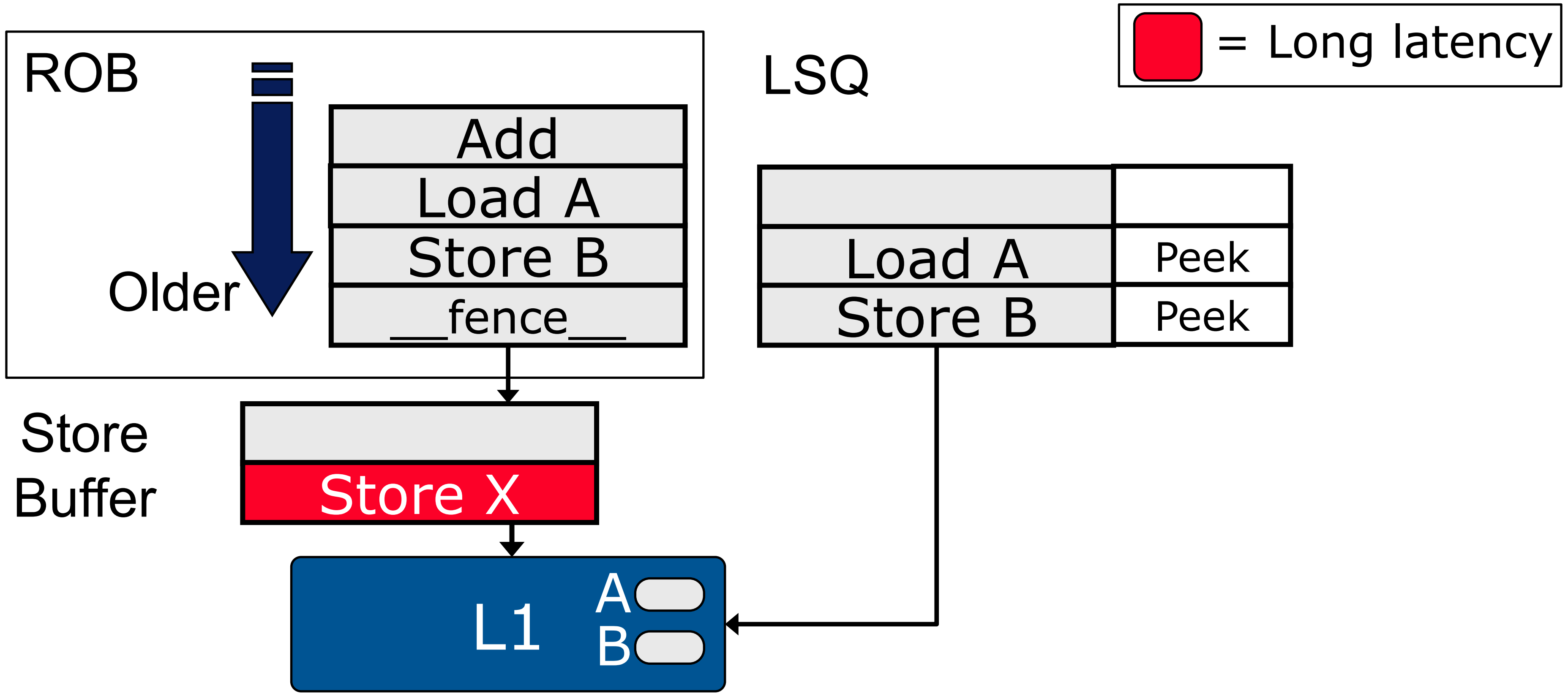


- Store B and Store A retired, and overtook Store X
  - All long latency operations happening in parallel

Example w. Weak Consistency and Fence

```
Store X        ; misses in L1 & L2
__fence__
Store B        ; hits in L1
Load A         ; hits in L1
Add
Store A
Load Y         ; misses in L1 & L2
Store Z        ; misses in L1 & L2
```

Execution with WC + Fence



- When fence reaches head of ROB, the CPU blocks
  - Wait for SB to drain and Store X to finish

Summary

- Cache coherence is not memory consistency
  - Memory models dictate what behavior can be observed across different memory locations
- Uniprocessor memory ordering
  - Ensure program order and access atomicity
  - Use a Load Store Queue to solve address overlap
- Basic consistency models: SC, PC, Weak
- Consistency is a very meticulous topic, and mistakes are often made!