Cache Coherence vs. Memory Consistency

- Cache coherence
  - deals with ordering of writes to a **single** memory location
  - only needed for systems with caches
- Memory consistency
  - deals with ordering of reads/writes to *all* memory locations
  - needed in systems with or without caches

Why is a memory consistency model needed?

[§9.1] Programmer's intuition:

P0:	P1:
S1: datum = 5;	S3: while (!datumIsReady);
S2: datumIsReady = 1;	S4: … = datum

Programmers expect **S4** to read the new value of **datum** (i.e., **5**).

This expectation is violated if—

- **S2** appears to be executed before **S1**
- **S4** appears to be executed before **S3**

Thus, *Hypothesis 1: Program-order expectation*

Programmers expect memory accesses in a thread to be executed in the same order in which they occur in the source code.

Not only the executing thread, but *all* threads, are expected to see them in this order.

P0:	P1:	P2:
S1: x = 5; S2: xReady = 1;	S3: while (!xReady) {}; S4: y = x + 4; S5: xyReady = 1;	S6: while (!xyReady) {}; S7: z = x * y;

Let's say, initially, **x = y = z = xReady = xyReady = 0**

As a programmer, what would you expect to be the value of **z** at **S7**?

This implies that if the new value of **x** has been propagated to **P2**, it has also been propagated to _____

Thus, *Hypothesis 2: Atomicity expectation*

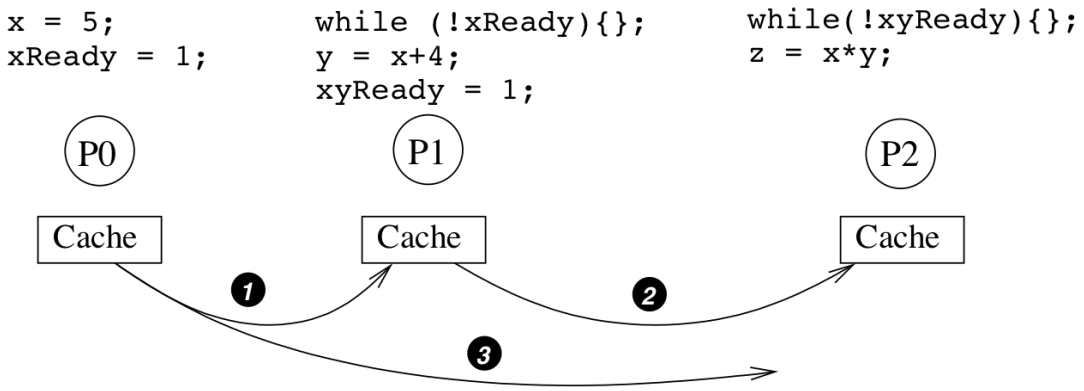
A read or write happens instantaneously with respect to all processors.

How can the atomicity expectation be violated?

Step 1: New values of **x** and **xReady** have been propagated to **P1**, but have not reached **P2**.

Step 2: New values of **y** and **xyReady** have been propagated to **P2** before **x** is propagated to **P2**.

Step 3: When **x** is propagated to **P2**, **P2** has already read the old value of **x**, and **z** has been set to **0**.



Is there any other way that a violation of store atomicity can lead to a wrong value for **z**?

What is *another* "incorrect" value that could be written for **z**? Explain how this could happen.

Summary of programmer's expectations:

Memory accesses emanating from a processor should be performed in program order, and each of them should be performed atomically.

These expectations were incorporated in Lamport's 1979 definition of sequential consistency:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Sequentially consistent vs. non-SC outcomes

Consider these code sequences, with **a** and **b** initialized to **0**.

P0:	P1:
S1: a = 1; S2: b = 1;	S3: print b; S4: print a;

Note that this program is *non-deterministic* due to a lack of synchronization.

Under SC, **S1** → **S2** and **S3** → **S4** are guaranteed

Assuming SC, what values might possibly be printed for **a** and **b**?

What values for **a**, **b** are impossible?

Prove it.

For **a** to print as ___, it must be that **S4** → **S1**: e.g.,

For **b** to print as ___, it must be that **S2** → **S3**: e.g.,

Both of these conditions cannot hold. Prove it.

On a non-SC machine, the outcome of **a**, **b** = ___, ____is possible. What statement ordering can produce it?

In this case, which of the two SC precedence guarantees (above) is violated?

Let's take another example.

P0:	P1:
S1: a = 1; S2: print b;	S3: b = 1; S4: print a;

Exercise: Assuming that **a** and **b** are initialized to 0,

- what values can be printed under SC?
- what values are impossible to print under SC?
- prove that the impossible results can only occur if SC is violated.

Answer: Note that the program is non-deterministic due to a lack of synchronization.

With SC, **S1** → **S2** and **S3** → **S4** are guaranteed

On a nondeterministic machine, the outcome **a**, **b** _____ is possible.

- **`S4`**, **`S1`**, **`S2`**, **`S3`**
    - In this case, **`S3`** → **`S4`** is violated
- **`S2`**, **`S3`**, **`S4`**, **`S1`**
    - In this case, **`S1`** → **`S2`** is violated

Both of the previous examples are non-deterministic.

Non-deterministic codes are notoriously hard to debug.

But non-determinism may have legitimate uses. See Code 3.16 (ocean-current simulation) and 3.18 (smoothing filter for grayscale image).

So, does preserving ordering of memory accesses matter?

- Probably not if non-determinism is intentional
- Otherwise, yes, because:
    - Helps keep programmers sane during debugging.
    - Even properly synchronized programs need ordering for the synchronization to work properly.

Building a SC system

[§9.2] Which of the two hypotheses (expectations) can be guaranteed by software?

- Ensure that compiler does not reorder memory accesses;
- Declare critical variables as volatile (to avoid register allocation, code elimination, etc.)

What hypothesis needs to be maintained by hardware?

- Execute one memory access one at a time, in program order. One access needs to be complete before the next can start.

- In the processor pipeline, memory accesses can be overlapped or reordered.
    - But they must go to the cache in program order.
    - A load is complete when the block has been read from the cache.
    - A store is complete when an invalidation has been posted (on a bus) or acknowledged (see details in §10.2.1).

Example of SC Ordering

```
S1: ld R1, A
S2: ld R2, B
S3: st R3, C
S4: st R4, D
S5: ld R5, D
```

S1 must complete before **S2**, **S2** before **S3**, etc.

Implications

- If **S1** is a cache miss but **S2** is a cache hit, **S2** still must wait until **S1** is completed. Same with **S3** and **S4**.
- **S4** must wait for **S3** to complete, even though stores are often retired early.
- **S5** must wait for **S4** to complete, even though they are to the same location!

Improving SC performance

Via prefetching

We still have to obey ordering, but we can make each load/store complete faster, e.g. by converting cache misses into cache hits:

- Employ load prefetching
    - As soon as address is known/predictable,
        - fetch before previous loads have completed,
    - issue a prefetch request to fetch the block in Exclusive/Shared state

- Employ store prefetching
    - As soon as address is known/predictable, issue a prefetch request to fetch the block in Modified state

But this is not a perfect strategy. Why not?

- Prefetch too late ⇒
- Prefetch too early ⇒

Via speculation

We can violate ordering, but undo the effect if atomicity is violated.

- The ability to undo execution and re-execute is already present in out-of-order processors (as covered in ECE 563).
    - So, we only need to determine when atomicity has been violated.
- Consider load A, followed by load B
    - In strict SC, load B must wait until load A completes
    - With speculation, load B accesses the cache anyway; the processor just marks load B as speculative
    - If B is invalidated before it “retires,” atomicity has been violated.
    - In this case, the architecture cancels B and re-executes it.

Store speculation is harder, because stores cannot be canceled. Hence, only load speculation is employed.