Example. D, F, R1, & R2 initialized to 0.

Program Order:

Core 1:	Core 2:
SW 1->D	LW F->R1
SW 1->F	LW D->R2

Possible outcomes:
- R1 = 0, R2 = 0 (Both LWs occur before the first SW)
- R1 = 1, R2 = 1 (Both LWs occur after the last SW)
- R1 = 0, R2 = 1 (Interleaved)

IMPOSSIBLE IN PROGRAM ORDER: R1 = 1, R2 = 0

Potential Execution Order in OOO Processor:

Core 1:	Core 2:	
SW 1->D	LW D->R2	REORDERED
SW 1->F	LW F->R1	

Possible outcomes:
- R1 = 0, R2 = 0 (Both LW occur before first SW)
- R1 = 1, R2 = 1 (Both LW occur after last SW)
- R1 = 0, R2 = 1 (Both LWs occur in-between the SWs)
- R1 = 1, R2 = 0 (Both SWs occur in-between the LWs)

Memory Consistency

Memory consistency determines the order of accesses by different cores. Memory consistency and sequential consistency go hand-in-hand in ensuring programs execute properly. Without sequential consistency a program execution cannot be guaranteed to have the same outcome with time. Memory consistency determines how strictly should we enforce ordering among accesses to different memory locations. This is needed to get what we expect from our Synchronization and Data Accesses in a Shared-Memory Program.

Memory Consistency

Coherence defines the order of accesses of threads to the same address. BY MULTIPLE CORES It does not say anything about accesses to different addresses. Coherence is needed to CORRECTLY share the SAME data among different threads.

Memory Consistency defines the order of accesses to different addresses. BY MULTIPLE CORES

Consistency Matters

Consistency ensures the program order is the same as the execution order. especially in OOO processors. Remember, Out Of Order processors are totally fine to Re-Order the execution of some INDEPENDENT instructions (different from the original order they were specified in the program) IF THEY'RE SEENS AS INDEPEDENT FROM THE CORE'S PERSPECTIVE. For ex, a LOAD A followed by a LOAD B in a program can be flipped in an OOO processor so that the LOAD B occurs before LOAD A, because they're independent. This would be fine on a Uniprocessor, however there can be issues when this occurs on Multi-Cores.

Why We Need Consistency

Additional ordering restrictions are needed to prevent the incorrect outcomes that can occur in Data Ready Flag synchronization and Thread termination.

Sequential Consistency

The result of any execution should be:

- As if the accesses were executed in-order by each processor (individually)
- As if the accesses among different processors were arbitrarily interleaved

Simplest Implementation of Sequential Consistency:

A core performs the next access only when all previous accesses are complete. Unfortunately this leads to poor performance.

In the following example, with Sequential Consistency, any core cannot access/read "data" until the read of the "flag" has completed/COMMITED.

```
while (!flag) {};
print data;
```

We can still have branch-pred, but we'd delay reading the data until the flag read has completed.

Better Implementation of Sequential Consistency

- A core can reorder loads
- Detect when sequential consistency may be violated and fix it.

This implementation would allow reordering of LOADs, but when it DOES reorder a LOAD, it is aware that it has done so, and needs to monitor the MEMORY BUS to see if any other cores are updating that same memory location with a STORE. If it doesn't see any updates, then it knows that the value has stayed the same and it is safe to commit. It listens to the bus this UNTIL the COMMIT pointer of the ROB reaches the COMMIT point of the reordered LOAD instruction. If it SNOOPed an update from another core for that MemLoc, then it knows it needs to scratch out that value and redo the LOAD. This produces correct, sequentially consistent behavior.

To detect and fix a SC violation:

~~The~~ monitor the coherence traffic produced by other processors. Coherence traffic refers to the load and store commands, especially out of order commands.

See top of next page for the 4 types of Ordering. Sequential Consistency will obey all 4 types of ordering, while Relaxed Consistency will only obey some of them (usually will not obey RD->RD but will obey the other 3).

Relax Consistency

Tell the programmers that they cannot expect sequential consistency for all instances. Usually RD A → RD B must be programmed in correct order.

Explanation of "Data-Ready Flag" Synchronization:
So the example shown in the lectures illustrate a case with a While Loop that keeps looping until an exit condition is met, and then after it is met, it accesses some DATA that it expects to be "ready" to be accessed. It's possible (without Consistency) for the core to PREDICT that it should exit the loop, so it will go ahead and access the DATA. However, at that EXACT moment the prediction could be incorrect, however it won't know that until a few cycles later when the actual Prediction outcome is resolved (load something from memory into a register and then compare it to the loop-exit-condition). Bc of branchpred, it goes ahead and allows the LOAD instruction of the DATA to enter the pipeline as well (allows it to enter the Instruction Queue). However, due to REORDERING WITHIN THE PIPELINE, the LOAD of the DATA could actually occur before the LOAD of the EXIT CONDITION (bc the processor is not aware of the "while loop", all it sees is a stream of instructions to process. From its perspective, the two loads are independent, and are allowed to be reordered IN TERMS OF EXECUTION. COMMIT OF THE DATA STILL HAPPENS IN ORDER). So it loads the DATA first (which is actually NOT READY TO BE ACCESSED), and then its possible that when it actually does the load/checking of the "loop-exit-condition", the other core has actually reach a point where it updates the DATA and sets the flag to indicate that it (the DATA) is ready to be accessed. So the first core sees that the exit condition is resolved and thinks that its prediction was correct, so it COMMITS the previously fetched data (which was just UPDATED by the second core), which is actually INCORRECT.

4 Types of Ordering:
- WR A -> WR B (keeping consecutive writes to different locations consistent)
- WR A -> RD B (keeping a read to location B after a write to A consistent)
- RD A -> WR B (keeping a write to B after a read from A consistent)
- RD A -> RD B (keeping a read of B after a read of A consistent)

So with Relaxed Consistency, the programmer must remember that two separate reads of different memory locations can be re-ordered relative to each other. However, the x86 architecture provides an "MSYNC" instruction that allows the programmer to explicitly specify to the processor a synchronization point so that ALL MEM ACCESS instructions AFTER (in program order) that point (the MSYNC instruction) can only occur (execution order) AFTER ALL the MEM ACCESS instructions that appear before that point in the program. So REORDERING OF MEM ACCESS IS ALLOWED BUT NOT ACROSS THE MSYNC INSTRUCTION.

Data Races and Consistency

Data race: accesses to the same address by different cores that are not ordered by synchronization.

A data-race-free program cannot create data races. A key property is the program will behave the same in any consistency model. So the program can be debugged in a sequential consistency model, then run on a relaxed consistency model.

Why? Because if your synchronization is implemented correctly, then your synchronization is going to create the right orderings for mutli-core, same-memory-location access, and then there is no opportunity for the Re-Orderings that might violate Sequential Consistencies. A Data-Race-Free-Program is actually Sequentially Consistent, even if we run it on a machine that can only provide some form of RELAXED Consistency model. Once we're sure that a program is Data-Race-Free, then we can relax the Consistency Model for improved performance (due to Re-Ordering) while keeping a peace of mind that the behavior will be deterministic.

In a non-data-race-free program anything can happen.

Consistency Models

-Sequential Consistency

-Relaxed Consistency Models There are several Relaxed Consistency Models ("Weak", "Processor", "Release", "Lazy Release", "Scope", etc...)

-The key to these -- all of them support synchronization operations that ensure the correct order occurs.

The key to all of these different types of models is this: Even though they all allow arbitrary re-orderings or maybe some re-orderings among data/memory operations, ALL of them support synchronization operations that allow you to ensure that the ordering that you really want in your program gets to happen. Some of them do it with something like the MSYNC operation, some using more refined memory synchronization operations, etc.

Remember that this is keeping memory access consistency due to multiple access by multiple cores. With a uniprocessor, all independent accesses can be reordered while still being correct. However, with multi-processors, what may seem as permissible reordering of MemAccess instructions could lead to inconsistencies due to the behavior of other processors. This is why, with multi-processors, there are stricter restrictions, or more considerations that need to be made when it comes to the Re-Ordering of Memory Access instructions than compared to uniprocessors.

SEQUENTIAL consistency ensures that all memory accesses execute as if we process them one at a time, each time selecting a core and letting it complete its next access in program order. Note that this does not require round-robin selection of cores - it even allows one core to be selected several times in a row. This allows for many possible interleaving's of accesses from different cores, but it PREVENTS ACCESSES FROM ONE CORE FROM BEING REORDERED.

WEAK consistency distinguishes between synchronization and non-synchronization accesses. Synchronization accesses (such as those used to ACQUIRE or RELEASE a lock) are NEVER REORDERED AMONGST THEMSELVES OR WITH OTHER ACCESSES. This means that synchronization accesses are done in a sequentially-consistent way, but NON-SYNCHRONIZATION ACCESSES that a core makes BETWEEN SYNCHRONIZATION ACCESSES can be reordered freely (note that this reordering is still limited by coherence and dependences in program order).

RELEASE consistency further distinguishes between ACQUIRE (e.g. lock acquire) and RELEASE (e.g. lock release) SYNCHRONIZATION ACCESSES. They are still not reordered amongst themselves, but non-synchronization accesses can be reordered freely, except that:
- READS/LOADS THAT OCCUR AFTER AN ACQUIRE EVENT (IN PROGRAM ORDER) CANNOT BE REORDERED TO OCCUR BEFORE IT.
- NON-SYNCHRONIZATION WRITES/STORES THAT OCCUR BEFORE A RELEASE SYNCHRONIZATION EVENT (IN PROGRAM ORDER) CANNOT BE REORDERED TO OCCUR AFTER IT.

Note that WEAK consistency can be achieved using RELEASE consistency by treating EVERY SYNCHRONIZATION EVENT as BOTH an ACQUIRE and a RELEASE. Also note that SEQUENTIAL consistency can be achieved using WEAK consistency by treating EVERY ACCESS as a SYNCHRONIZATION ACCESS.

MOST STRICT -----> MOST RELAXED
SEQUENTIAL > WEAK > RELEASE

With Release Consistency, a write that sits before an Acquire event (in program order) could possibly be reordered to executed AFTER the acquire event. Nothing restricts this.

So whatever consistency model we choose will AFFECT THE REORDERING POTENTIAL OF ALL INSTRUCTIONS IN A PROGRAM, NOT JUST CRITICAL SECTIONS. So selecting SEQUENTIAL consistency will prevent reordering of ALL instructions for all the cores, which will greatly diminish the performance of the system for the program. That's why it's so important to ensure that your SYNCHRONIZATION is correct. If it is correct, you can take advantage of the performance benefits that an Out Of Order processor provides while still ensuring that the shared-memory access portions occur correctly as the programmer expects.