# Memory Consistency

## Introduction

1. How strictly should we enforce ordering of accesses to different memory locations?
   - Necessary for handling synchronization in a shared memory program

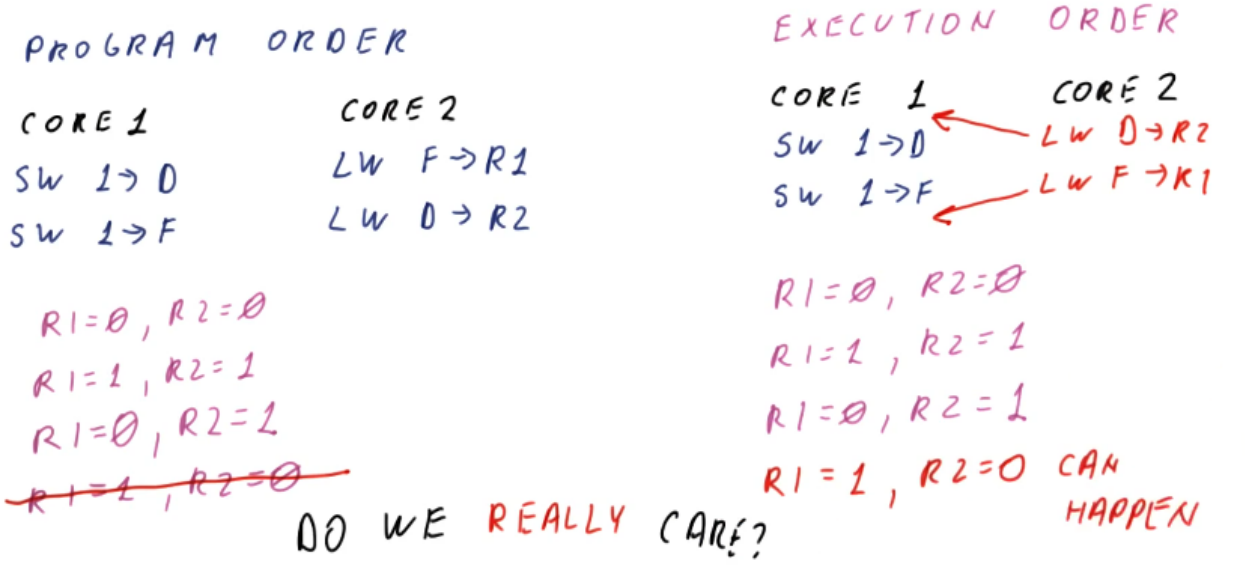
## Memory Consistency

1. Coherence: Defines order of accesses to the same address
   - We need this to share data
   - Does not say anything about accesses to different locations
2. Consistency: Defines order of accesses to different addresses
   - Does this even matter?

## Consistency Matters

1. If one core stores 1 to memory locations A and B (zero initially) and another core reads the values from A and B
   - If we reorder the loads (program order != execution order) then the loads could be interleaved with stores
   - This means that, if we aren't careful, we can end in an invalid state according to program order, but not according to execution order
2. This does not violate coherence at all

---




---

Example

---

## Consistency Matters Quiz

1. Coherent memory, out-of-order cores
   - Branches can be predicted
   - Stores occur strictly in program order
   - Loads can be reordered
   - Initially, flag = 0 and data = 0
2. Core 1:

```
while(!flag) {
    wait;
}
printf(data);
```

3. Core 2:

```
data = 10;
data = data + 5;
flag = 1;
```

4. What values could be printed?
   - 0 (yes)
   - 5 (no; stores can't be reordered)
   - 10 (yes)
   - 15 (yes)
   - Other (no; stores can't be reordered)
5. Coherence can't prevent this execution; we need consistency
   - The branch predictor will predict that we leave the loop correctly, but the writes may not have finished, so data could be 0, 10, or 15

## Why We Need Consistency

1. Data-ready flag synchronization
   - Previous quiz
2. Thread termination
   - Thread A creates thread B
   - Thread B works, updates data
   - Thread A waits until B exits (system call)
   - Thread B done, OS marks B done
3. Keep additional ordering restrictions beyond coherence alone

## Sequential Consistency

1. Sequential consistency: The result of any execution should be as if accesses executed by each processor were executed in-order and accesses among different processors were arbitrarily interleaved
2. Simplest implementation: A core performs the next access only when all previous accesses are complete
   - Can still branch predict, but must delay until accesses are complete
   - This works well, but is bad for performance

## Simple Implementation of SC Quiz

1. Simple implementation: A core performs the next access only when all previous accesses are complete
2. What is the memory-level parallelism we get on each core?
   - 1; this forces all memory accesses to occur in serial

## Better Implementation of SC

1. Core can reorder loads
   - Must detect when SC may be violated and fix things
   - How do we do this?
2. Violations of SC occur when we read something ahead of program order and another instruction writes to that memory location
   - Need to monitor coherence traffic from other cores
   - If this happens, replay the load instruction
     - This is fine since it hasn't been committed from the ROB yet

## Relaxed Consistency

1. Alternative to sequential consistency
2. Four types of ordering
   - WR A -> WR B
   - WR A -> RD B
   - RD A -> WR B
   - RD A -> RD B
3. Sequential constistency means all orderings must be obeyed
4. Relaxed: Some orderings are not obeyed (RD -> RD for example)
   - How do we write correct programs if the processor doesn't guarantee this ordering?
5. Relaxed consistency: Allow reordering of "normal" accesses
   - Add special non-reorderable accesses
     - Must use these when ordering matters
   - Example: x86 MSYNC instruction
     - Normally, all accesses must be reordered
     - But no reording across MSYNC
   - Allows for optimization where reordering is permitted while still maintaining correctness from the view of the programmer

```
while(!flag);
MSYNC
printf(data);
```

## MSYNC Quiz

1. All 4 reorderings are allowed
2. Where do we need an MSYNC call in the following program?

```
WAIT: LL R1, lock
BNEZ R1, WAIT
SC R2, lock
BEQZ R2, WAIT
MSYNC
LW VAR
INC
SW VAR
MSYNC
SW 0, lock
```

3. Lock is considered an "acquire"
   - MSYNC after
4. Unlock is considered a "release"
   - MSYNC before

## Data Races and Consistency

1. Data race: When RD -> WR, WR -> RD, or WR -> WR between accesses on different cores that are not ordered by synchronization
2. Data-race-free program: Program cannot produce any data races
   - Behave the same as SC in any consistency model
3. Non data-race-free programs -> anything can happen
   - Some processors allow users to flip between sequential consistency and other, more relaxed consistency models that provide better performance

## Consistency Models

1. Sequential consistency
2. Relaxed consistency: Violate one or more ordering rules
    - Weak
    - Processor
    - Release
    - Lazy release
    - Scope

## Conclusion

1. Memory consistency defines what is allowed to happen when processor cores are reordering memory accesses
2. Sequential consistency gives the expected behavior, but limits performance
3. More relaxed consistency models improve performance but can make program behavior more difficult to reason about