A conceptual schematic for a multicore processor is shown below.

To reduce the average memory access time, each of the four cores has its own cache, which will satisfy most memory requests.

If there's a cache miss, a request is sent to the shared main memory.

With a modest number of cores and a good cache hit ratio, the number of memory requests that must access main memory during normal operation should be pretty small.

To keep the number of memory accesses to a minimum, the caches implement a write-back strategy, where ST instructions update the cache, but main memory is only updated when a dirty cache line is replaced.

Our goal is that each core should share the contents of main memory, i.e., changes made by one core should visible to all the other cores.

In the example shown here, core 0 is running Thread A and core 1 is running Thread B.

Both threads reference two shared memory locations holding the values for the variables X and Y.

The current values of X and Y are 1 and 2, respectively.

Those values are held in main memory as well as being cached by each core.

What happens when the threads are executed?

Each thread executes independently, updating its cache during stores to X and Y.

For any possible execution order, either concurrent or sequential, the result is the same: Thread A prints "2", Thread B prints "1".

Hardware engineers would point to the consistent outcomes and declare victory!

But closer examination of the final system state reveals some problems.

After execution is complete, the two cores disagree on the values of X and Y.

Threads running on core 0 will see X=3 and Y=2.

Threads running on core 1 will see X=1 and Y=4.

Because of the caches, the system isn't behaving as if there's a single shared memory.

On the other hand, we can't eliminate the caches since that would cause the average memory access time to skyrocket, ruining any hoped-for performance improvement from using multiple cores.

What outcome should we expect?

One plausible standard of correctness is the outcome when the threads are run a single timeshared core.

The argument would be that a multicore implementation should produce the same outcome but more quickly, with parallel execution replacing timesharing.

The table shows the possible results of the timesharing experiment, where the outcome depends on the order in which the statements are executed.

Programmers will understand that there is more than one possible outcome and know that they would have to impose additional constraints on execution order, say, using semaphores, if they wanted a specific outcome.

Notice that the multicore outcome of 2,1 doesn't appear anywhere on the list of possible outcomes from sequential timeshared execution.

The notion that executing N threads in parallel should correspond to some interleaved execution of those threads on a single core is called "sequential consistency".

If multicore systems implement sequential consistency, then programmers can think of the systems as providing hardware-accelerated timesharing.

So, our simple multicore system fails on two accounts.

First, it doesn't correctly implement a shared memory since, as we've seen, it's possible for the two cores to disagree about the current value of a shared variable.

Second, as a consequence of the first problem, the system doesn't implement sequential consistency.

Clearly, we'll need to figure out a fix!

One possible fix is to give up on sequential consistency.

An alternative memory semantics is "weak consistency", which only requires that the memory operations from each thread appear to be performed in the order issued by that thread.

In other words, in a weakly consistent system, if a particular thread writes to X and then writes to Y, the possible outcomes from reads of X and Y by any thread would be one of (unchanged X, unchanged Y), or (changed X, unchanged Y), or (changed X, changed Y).

But no thread would see changed Y but unchanged X.

In a weakly consistent system, memory operations from other threads may overlap in arbitrary ways (not necessarily consistent with any sequential interleaving).

Note that our multicore cache system doesn't itself guarantee even weak consistency.

A thread that executes "write X; write Y" will update its local cache, but later cache replacements may cause the updated Y value to be written to main memory before the updated X value.

To implement weak consistency, the thread should be modified to "write X; communicate changes to all other processors; write Y".

In the next section, we'll discuss how to modify the caches to perform the required communication automatically.

Out-of-order cores have an extra complication since there's no guarantee that successive ST instructions will complete in the order they appeared in the program.

These architectures provide a BARRIER instruction that guarantees that memory operations before the BARRIER are completed before memory operation executed after the BARRIER.

There are many types of memory consistency - each commercially-available multicore system has its own particular guarantees about what happens when.

So the prudent programmer needs to read the ISA manual carefully to ensure that her program will do what she wants.

See the referenced PDF file for a very readable discussion about memory semantics in multicore systems.